STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LVII**, Number 2, 2012

DRAS: DERIVED REQUIREMENTS GENERATION

DAVID BAR-ON AND SHMUEL TYSZBEROWICZ

Abstract. A system specification may include many interdependencies between the specified requirements. Requirements may conflict with one another and they may impact other requirements as well. We present the DRAS (Derived Requirements generation by Actions and States) methodology that helps to identify Functional Requirements (FRs) that are in conflict with other FRs DRAS also assists with generating the derived requirements that are inferred from the conflicting requirements. DRAS is based on the observation that using the same action in two requirements indicates that those requirements may conflict. In order to find which requirements potentially are in conflict with a given requirement, DRAS considers actions stated by the requirements, their implied actions, modes (also called states), and action modifiers.

The DRAS methodology is explained by means of a comprehensive example, which uses a subset of simplified requirements from an industrial project that one of the authors participated in its requirements definition and analysis.

1. Introduction

System and product requirements often contain inconsistent and conflicting requirements. Requirements may impact other requirements as well. Inconsistency between requirements usually means that either existing requirements should be enhanced, or new requirements should be written. The DRAS methodology (***DRAS*** — *Derived Requirements generation by Actions and States*) that is described in this paper helps to identify conflicting Functional Requirements (FRs) and to resolve the conflicts. The method is not effective for Non-Functional Requirements (NFRs).

Consider the functional requirements "close the window when the outside temperature is below 10 degrees" and "open the window in the morning".

Received by the editors: April 25, 2012.

1991 *Mathematics Subject Classification.* 68N99.

1998 *CR Categories and Descriptors.* code [**D.2.1**]: Requirements/Specifications – *Methodologies* ;

Key words and phrases. Early Aspects, Requirements Engineering, Conflicting Requirements, Crosscutting Requirements, Derived Requirements, Match-Point.

The requirement to close the window conflicts with the requirement to open the window when the temperature in the morning decreases below 10 degrees. The outcome of analyzing the interaction between the requirements should include a decision whether or not to open the window in the morning, when the temperature is below 10 degrees.

The resolution of conflicting requirements influences the definition and the architecture of the system. Therefore, it is very important to identify conflicting requirements as soon as possible in the software development process, for example during requirements analysis. Otherwise there may be a major overhead work during later development phases. While identifying and handling conflicting requirements, both functional-requirements and non-functional requirements should be considered. A rigorous analysis and understanding of conflicting requirements and their interactions is essential to derive a balanced architecture. Ignoring it may result in an incomplete understanding of specified requirements and, consequently, poorly informed architectural choices.

One way to resolve the conflicts between requirements is to define Derived Requirements (DRs). These are requirements that we infer, or derive, from other user requirements. They represent the outcome of resolving interactions and conflicts between requirements. DRs may be either new requirements or enhancements to existing requirements.

The identification of conflicting requirements is a difficult process, especially in large systems; consequently, methods and tools that can identify conflicting requirements and define the resulting DRs are needed. This is because the system may be designed according to one of the requirements, while the other conflicting requirements should have precedence, so a redesign may be required later in the process.

The DRAS methodology is used to generate textual DRs from stakeholders' textual requirements. DRAS aims to assist the process of identifying conflicting functional requirements and supports resolving these conflicts. The resolution may include changing, enhancing, or overriding some of the conflicting FRs, including the generation of textual derived requirements which are part of the resolution. Note that DRAS does not handle non-functional requirements, as it relies on action words to identify conflicts between requirements.

We have addressed the main concepts of DRAS methodology in [5, 4]. This paper expands on the previous papers by providing more details about the methodology, procedures used for its implementation (using a prototype tool), and adding an Appendix with a comprehensive example of the DRAS process.

DRAS includes some ideas that have been adopted from existing methods for identifying and handling conflicting requirements, especially from [2, 31,

7]. DRAS uses actions as the primary means for identifying conflicting FRs, similar to ý [2]. Actions are the functions specified by the FRs. In the example above, "open window" is the action used by both requirements. Note that using the same action by both requirements indicates that one of them may conflict with the other.

The main contribution of the DRAS methodology is the way conflicting requirements are identified. In order to find which requirements may conflict a given requirement *Req*, DRAS considers *Req's actions, modes* (also called states), and *action modifiers*. Actions can be explicitly stated in *Req*, implied by actions in *Req*, or imply actions in *Req*. For example, "refresh the room"implies the action "open a window" and "unlock the window" is implied by "open the window" (and indirectly, is also implied by "refresh the room").

The activation of some actions depends on the mode of the system *Req* refers to. For example, there might be different requirements for opening the window depending on the season - summer or winter. Requirements that are relevant only for the summer usually are not conflicting requirements that are relevant only for the winter.

We also distinguish between a restricting action, that forbids some activities (e.g., "open a window" is restricted in "do not open the window when the temperature is below 10 degrees") and ease a restriction for an action (e.g., "the window may be opened when leaving the room, even if the temperature is below 10 degrees"). This distinction is used to determine whether to consider implied or implying actions.

In designing a system that reengineers the requirements, one dilemma is whether the output should be expressed textually or a more formal method should be used. The initial stakeholders' requirements for a system or a product are usually textual. Only later in the development process are the requirements specifications transformed into a more formal, technical representation (such as UML diagrams ý [16]). When the data is formally represented, it is easier to identify conflicting requirements and the requirements they conflict. However, non-technical stakeholders, such as customers and marketing representatives, usually are not trained to read formal specifications. Therefore, it is advantageous to be able to generate DRs in a textual form and to integrate them with other requirements. This enables non-technical stakeholders to review and understand the specifications. Therefore we decided to create textual requirements as the output of DRAS.

In order to generate textual DRs from stakeholders' requirements, DRAS first identifies *match-points* (defined in ý [7]) between requirements. A *Match-point* in requirements is identifying tentative conflicts between the requirements (e.g., a common action). This is performed by identifying common

actions that are used by the requirements (inspired by ý [2]), and by identifying common system modes and states (when these actions are used). The DRs are then created based on the conflicting requirements. This enables review and evaluation process by both technical and non-technical stakeholders.

As an example, following is a simple set of requirements for initiating a call from a cellular system:

- *Ra. When a phone user dials a number, the phone shall initiate a call to the dialed number.*
- *Rb. The phone shall allow initiating calls to the police (911 in the US, 112 in Europe) under any condition.*
- *Rc. The phone shall be allowed to initiate calls and receive calls only after checking that the user is allowed to use it (bills paid, phone not stolen, etc.).*

The first observation is that all of these requirements are FRs, and the action "call initiation" is mentioned in each of them. Therefore, the requirements may conflict with each other. Further analysis reveals that *Rb* and *Rc* are tentatively conflicting, as *Rc* restricts call initiation while *Rb* eases restriction for initiating a call. Analyzing each pair of requirements shows that *Rc* conflicts *Ra*, restricting its specification. Assuming that *Rc* has a higher priority than *Ra*, the result is an enhancement (change) to *Ra*. The enhanced derived requirement may be:

- *Rd. When a user of a phone dials a number, the phone shall initiate a call to the dialed number only if the user is legitimate.*

In this case *Rc* is redundant since *Rd* includes it. It still may be important to keep such a requirement, as usually not all the requirements it conflicts are identified in the early stages of development.

Another derived requirement is required since *Rb* and *Rc* are in conflict with each other. Illegitimate users are allowed to dial the police according to *Rb* and disallowed according to *Rc*. A common solution in cellular systems gives *R2* higher priority, allowing also illegitimate users to dial the police:

- *Re. Illegitimate user should be allowed to dial the police.*

Alternatively, this can be an enhancement to *Rd*:

- *Rd (enhanced). The phone should not allow dialing by an illegitimate user, unless the user dials the police.*

This paper is structured as follows. Section 2 gives an overview of related work. In Section 3 we discuss the problem of generating DRs and outline how DRAS handles this issue. Section 4 concludes the paper, and discusses

further work and enhancements to DRAS. The Appendix contains a detailed description of DRAS, which is demonstrated using detailed examples.

2. Related Work

Many methods exist to identify crosscutting and conflicting requirements. Since DRAS uses ideas of Aspect Oriented Requirements Engineering (AORE), we reference AORE-based methods for identifying crosscutting concerns, and use some of those ideas to identify conflicting requirements.

AORE complements existing requirements engineering approaches by offering additional abstraction and composition mechanisms for systematically handling crosscutting, or aspectual, requirements. AORE methods include techniques for explicitly modeling aspects or concerns, in the context of requirements specifications. These methods allow identifying concerns in requirements, identifying crosscutting aspects between requirements, evaluating the aspects, and resolving conflicts caused by aspectual requirements. An extensive review of AORE methods can be found in [5, 1].

Several of the referenced papers are using the term ***crosscutting requirements***, referring to requirements that are inconsistent with other requirements they crosscut, requirements that enhance or change the functionality of other requirements, and sometimes are also in conflict with these requirements. The DRAS methodology mainly deals with ***conflicting requirements***. However, several of the methods used for handling crosscutting requirements are also applicable for conflicting requirements and are adopted by DRAS.

Section ǯ″2.1 elaborates on AORE methodologies that includes ideas that are used DRAS. We describe some of the methodologies that have been evaluated during the development of DRAS in Section ǯ″2.2, including a brief discussion on their relevancy to DRAS.

2.1. Methods with ideas used by DRAS.

Goal Oriented Requirements Analysis [26] explores the alternatives for achieving the goals as described in a given set of high level requirements, in order to achieve high-level requirements and select the proper alternatives. The method correlates *Softgoals* (NFRs) with goals and other Softgoals, which is similar to analyzing crosscutting requirements. An enhancement of this method is ***Aspects in Requirements Goal Models*** (Yu et-al [36], summarized in [11]) proposes a process for discovering aspects early in the software development process, based on relationships between functional and non-functional goals. V-shape graphs are used to decompose goals from Softgoals, and to present the goal/softgoal hierarchies. The *Discovering Aspects from Requirements Goal Models* method described below is based on these methods.

Discovering Aspects from Requirements Goal Models method [36] is an enhancement of the Goal Oriented Requirements Analysis method. It proposes a process for discovering aspects early in the software development process, from relationships between functional and non-functional goals. DRAS enhancements over this method include correlating between system specifications to identify conflicting functional requirements and then resolving those conflicts.

Modularization and Composition of the Aspectual Requirements method, described in [31], defines an AORE process model for resolving conflicts between requirements, and determining the influence of the conflicts resolution later in the architecture and design development stages. It focuses on the modularization and on the composition of requirements-level concerns that cut across other requirements. The method is primarily useful for Non Functional Requirements (NFRs), such as: availability, security, response time, accuracy, reliability, and other requirements that cannot be encapsulated by a single viewpoint or use-case [18]. The method supports the separation of crosscutting NFR properties and helps to identify the mapping and influence of the requirements level aspects on artifacts at later development phases. The method identifies critical tradeoffs before the architecture is derived. It includes steps for identifying concerns and viewpoints relationships, identifying candidate aspects, identifying match-points, defining composition rules, and the composition of viewpoints and aspects using composition rules. DRAS adopted the same idea from the *Modularization and Composition of Aspectual Requirements* method, for analyzing the requirements in order to find out whether they are conflicting or whether a requirement enhances other ones, and how. However, as noted earlier, this method is mainly relevant for crosscutting NFRs while DRAS handles FRs; it also does not use actions as a means to identify crosscutting and conflicting functional requirements.

Composition Process for Aspect Oriented Requirements *(AOR)*, described in [7], is a process for composing crosscutting concerns with the concerns (requirements) they conflict. The method is used primarily for NFRs, but it also includes techniques that are applicable for functional requirements. The process includes the following steps: a) identify concerns; b) identify candidate aspects; c) compose candidate aspects with concerns. DRAS adopted from this methodology the identification of match points and the use of contribution and composition-rules. Note that this method mainly handles NFRs; it does not work well to identify match-points between FRs. Brito, et al. [8] presents enhancements to this approach; the composition rules are refined with new operations inspired by LOTOS [6] operators.

Theme and ***Theme/Doc*** are defined in [2] and [3]. These methods handle FRs, rather than mainly NFRs as in most of the other methods described

earlier. *Theme* is a method and set of tools developed for early identification of aspects in the software development life cycle. The *theme* notion represents a system feature. Themes can be either *base themes,* which may share some behavior structure with other base themes, or *crosscutting themes-aspects*, which have a behavior that overlays the base themes functionality. *Theme/Doc* can identify aspects from interrelated behaviors of FRs, not just from NFRs as most other methods identify. The *Theme/Doc* approach and tool is used to view the relationship between behaviors in requirements documents and to identify and isolate aspects in the requirements. That is, it is primarily used to identify and separate aspects or concerns from the requirements, but not to combine aspects with other requirements. The approach provides requirements specifications views, exposing relationships between behaviors in the system. The method helps to determine which elements of functionality are *base* and which are *aspects*. The *Theme/Doc* approach assumes that if two behaviors are described in the same requirement, they are related. According to this approach, behaviors can be related to each other in three ways: *erroneously/coincidentally*, *hierarchically,* and *crosscutting*. The *Theme/Doc* process includes creating a list of actions that are used by the system. The actions are then used to identify crosscutting requirements, based on how these requirements use the actions. Those ideas were adopted by DRAS. *Theme/Doc*, however, does not take into account implied actions, actions that are the result of initiating other actions and states/modes related to the requirements. DRAS handles these issues and this enables identifying conflicts between requirements that are using related actions, but not using the same actions.

2.2. **Other Methods Evaluated.** ***Viewpoints*** [15] are used to specify the system from the perspectives (viewpoints) of each of its users (Actors in Use Case diagrams). Usually each perspective is partial and incomplete, because of the different roles each user has. A separate evaluation for each viewpoint is needed in order to define the full system's specification. For a complex system, using viewpoints allows Separation of Concerns (SoC) between different viewpoints, and provides a more manageable means of handling the system's specifications. Viewpoint-oriented methods do just that. Nuseibeh [28] presents a viewpoint as collecting and partitioning knowledge about representations, processes, and products of software development - all from the user's perspective. Several Viewpoints-Based Requirements Engineering (VBRE) methods exist. For example, Silva [33] introduces an approach for classifying and diagnosing discrepancies between viewpoints. The main purpose of Viewpoint methods is to verify that requirements cover all perspectives. Although they deal with

SoC, they do not specifically treat the issue of identifying crosscutting or conflicting requirements. Therefore, ideas from these methods were not used for DRAS (at least not directly).

Adaptation of the NFR-Framework to AORE [34] is an enhancement to the method defined in [31], which also includes some methods defined in [26], and to the NFR-Framework [12]. The NFR-framework is goal-oriented and is similar to the method described in [26], using the notion of *softgoals*. The *Adaptation of the NFR-framework to AORE* method uses the general AORE process framework defined by Rashid, et al. [31]. The main enhancement over [31] is identifying and selecting *operationalizations*. To *operationalize* a requirement means to provide more concrete and precise mechanisms (such as operations, processes, data representations, or constraints) to solve a problem. To *operationalize a softgoal* is to define possible solutions, or design alternatives, to help achieve the NFRs. This method is applicable to DRAS, but the related ideas used in DRAS were taken from [31] instead.

Goal Oriented Requirements Methodology [14] enhances the *Adaptation of the NFR-Framework to AORE* [34], and is based on the SoC principle. The method provides a way to represent crosscutting requirements separately from the requirements they affect and to specify the composition between them.

Crosscutting Quality Attributes [25, 9] propose a model to identify and specify *Quality Attributes* (QA) that crosscut other requirements at the requirements analysis stage. A QA is a non-functional concern, such as response time, accuracy, security, and reliability. This is the same as in a NFR, but from the point-of-view of the functional requirement. The aim of the method is to improve the separation of crosscutting requirements during analysis, providing a better means for identifying and managing conflicts. The QA method enables the handling of the NFR aspect of the FR together with the FR, instead of handling each of them separately. This method is only partly applicable for generating derived requirements from the requirements defined in this work. However, it mainly handles NFR and QA requirements. Also, the main methods it uses are also included in other methods [31]. Therefore, ideas from this method were not incorporated into DRAS.

3. The DRAS Methodology: Generating Derived Requirements

The DRAS methodology is used to identify and handle conflicting functional requirements. DRAS first identifies the actions used by each requirement, including implied actions, the modes (or states) that are relevant for the requirement, and the action modifiers per action. Based on this information, DRAS identifies the functional conflicting requirements, the requirements they

are in conflict with, and helps with generating the resulting *derived requirements (DRs)*. The generated requirements are textual, so that all stakeholders, including those with no technical background, can review and understand the requirements.

We have chosen to identify conflicting FRs based on the *actions* used by the requirements, similar to the Theme/Doc method [2, 3], as the functional requirements are used to specify the functions, or actions, that the system should provide.

3.1. **Input FRs.** Following are functional requirements that will be used to describe the DRAS process. The requirements are a simplified version of a very large project that one of the authors participated in the definition and the analysis of its requirements. The requirements are mainly about the Push-to-Talk (PTT [37]) action that is used to initiate calls to a pre-selected user or target number in walky-talkies, by pressing a button, also called PTT. As in walky-talkies, these calls are half-duplex, and only one participant can transmit voice at a given time.

- *R1. When the PTT is pressed, the phone shall initiate a call.*
- *R2. When another phone transmits, the phone shall not initiate voice transmission.*
- *R3. During a call, the phone shall not initiate a voice transmission when another phone transmits.*
- *R4. In the Emergency mode, the phone should always be allowed to initiate a voice transmission.*
- *R5. Illegitimate user should not be allowed to use the cellular system.*
- *R6. Illegitimate users shall not be allowed to initiate calls.*
- *R7. All users should be allowed to initiate a call to the police (an emergency number).*

As described in the introduction, DRAS purpose is to identify conflicting requirements and to generate derive requirements to resolve the conflict, based on the actions in the requirements and the relative requirements priorities.

For example, in this set of requirements *R1* and *R2* may conflict, since both are about *voice transmission*, as a call initiation implies voice transmission. In cellular systems, when the PTT is used to initiate a call, usually *R1 and R2 are conflicting requirements*. Since usually *R2* has a higher priority, i.e., a phone will not try to transmit if another phone already transmits, the conflict resolution may be as follows [the E in *R1(E)* means enhanced]:

- *R1(E). When PTT is pressed, the phone shall initiate a call unless another phone transmits.*

Note that with *R1(E)*, *R2* may be redundant. However, it is important to keep such conflicting requirements. Usually not all requirements that are in conflict with each other are identified in the early stages of development; moreover, new conflicting requirements may be added later (changes and additions to the systems' requirements usually never end).

3.2. **DRAS Outline.** The process map for the DRAS methodology is shown in Figure 1. The methodology steps are briefly described in the following sections, using as an example the requirements defined above. In the Appendix we provide a detailed example of using DRAS.

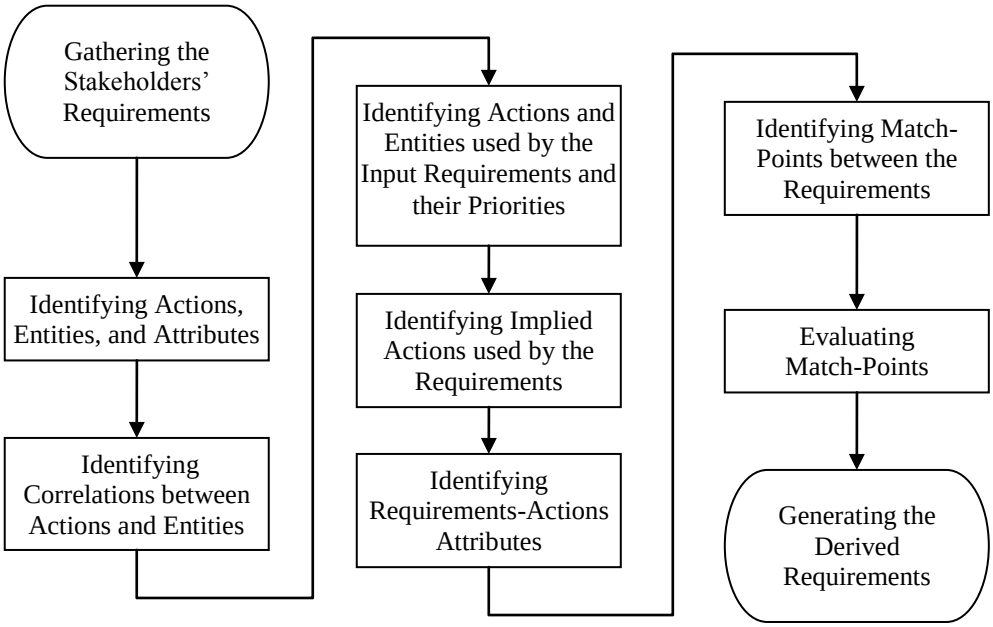

FIGURE 1. DRAS: Process Map

3.3. **Identifying Actions and Attributes Lists.** After the stakeholders' requirements are gathered and formulated, the lists of *actions*, *entities*, and *attributes* used in the system are identified. Attributes are mainly the *Modes* (and States) list of the system.

A list of contradicting pairs of modes values is also generated. This list is later used to remove tentatively identified conflicts between requirements that usually cannot occur in reality. E.g., a requirement related to call in process

usually cannot contradict with a requirement related to the case when the phone is not registered to the system.

In addition, correlations between actions and entities are identified, i.e. which actions are relevant for each entity. This is used to identify which actions are relevant to a requirement that refers to an entity, e.g. the actions related to *R5* that refers to the *cellular system* entity. However, in this section we will not refer to entities, and the relevant information can be found in the Appendix.

3.4. **Identifying Requirements and Actions Attributes.** The attributes assigned to the requirements and to the requirements' actions for assisting with the DRAS process are:

(1) Requirement's *Priority* (Section 3.4.1).
(2) Requirement's *Modes* (and *States*) (Section 3.4.2).
(3) Requirement's Action *Action Modifier* (Section 3.6.1).

Note that attributes assigned to a requirement are also assigned to the actions it uses (directly or indirectly).

3.4.1. *Requirements Priority.* The resolution of conflicting requirements depends on the relative priority of the requirements; i.e., deciding which of the conflicting requirements takes precedence. The specification of a requirement with a higher priority should override the specifications of requirements with a lower one. The use of relative priorities between requirements for handling conflicting requirements is inspired by methods such as [2, 31].

Note that it is time consuming to decide for each pair of requirements which requirement has a higher priority. In order to simplify the process, DRAS assigns a unique priority to each requirement. A functional requirement priority is based both on the importance of the actions and on the system-state referred to by the requirement. For example, a requirement about emergency actions will usually have higher priority than a requirement about other actions. Also, requirements that restrict operation in emergency states will usually have higher priority than requirements for general states.

The decision about a requirement priority is not always deterministic and the final decision is usually done (or at least validated) manually, based on experience, domain knowledge, understanding the customer needs, etc.

For example, the analysis of requirements priorities can be used to resolve the conflict between *R6* and *R7*:

- *R6. Illegitimate users shall not be allowed to initiate calls.*
- *R7. All users should be allowed to initiate a call to the police (an emergency number).*

The resolution whether an illegitimate user can dial the police or not, can only be performed manually. That is, we must determine which of these two requirements has a higher priority to define the proper DR.

It should be noted that assigning a unique priority per requirement is a simplification, as the requirements priorities are not necessarily totally ordered. That is, even if *Req A* has a higher priority than *Req B* and *Req B* has a higher priority than *Req C*, *Req C* still may have higher priority than *Req A*. Thus, using a unique priority per requirement can only suggest which requirement has a higher priority. A main reason for this is that many of the requirements are unrelated, so it difficult to compare their relative priority. Also, requirements may refer to more than one action, and each reference to an action may have its own priority.

For example:

- *R14. When pressing PTT, the phone shall initiate a call.*
- *R15. Illegitimate users shall not be allowed to transmit.*
- *R16. The phone shall send its location to the system every minute.*
- *R17. During a call, the phone shall not transmit its location.*

As initiating a call requires transmission, *R15* is assigned a higher priority than *R14*. However, although sending location to the system requires transmission as well, it still may be allowed for illegitimate users, e.g., to allow locating the phone in case of emergency. Therefore, *R16* is assigned a higher priority than *R15*, and thus also an higher priority than *R14*. However, due to *R17* (which is the result of a technical limitation of the system), initiating a call will stop sending the location for the duration of the call. If *R17* has a higher priority than *R16*, *R14* should also have higher priority than *R16* to allow initiating calls. Otherwise, *R17* has no meaning. Thus, different considerations lead to different relative priorities of *R14* and *R16*. Hence, the relative priorities between the requirements are not totally ordered.

3.4.2. *Requirements Modes and Conflicting Modes.* DRAS uses identified *modes* (and *states*) that each requirement refers to. Normally, when two requirements relate to two orthogonal modes, the requirements are not in conflict. That is, even if the two requirements use the same (implied) action, we can still assume that they do not have match-points. Examples for modes of a cellular phone are: a) is it in a call or not; b) is the user in the dialing a number; c) is the user reading SMS messages. As shown in Figure 2, the *Idle* and *Call* modes are orthogonal. A phone can either be in a call session or idle. Therefore a requirement that refers to the *Idle* mode will usually not conflict with a requirement referring to the *Call* mode.

However, the *Emergency* mode crosscuts both the *Idle* and the *Call* modes, since *Emergency* mode can be initiated no matter if the phone is in a call or not.

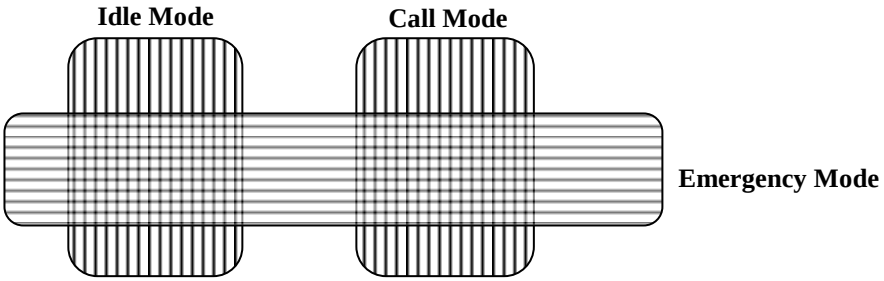

FIGURE 2. Crosscutting Modes

For example:

- *R3.* ***During a call****, the phone shall not initiate a* ***voice transmission*** *when another phone transmits.*
- *R4. In the* ***Emergency mode****, the phone should always be allowed to initiate a* ***voice transmission****.*

In this example both requirements use the same action (*transmit*), *R3* refers to the *Call* mode, and *R4* refers to the *Emergency* mode. Therefore, requirement *R4* tentatively conflicts with *R3*. Following is a possible solution to the conflict, assuming requirements related to emergency cases have higher priority than other requirements:

- *R3(E). In the Call mode, the phone shall not initiate a voice transmission when another phone transmits, unless it is in the Emergency mode.*

On the other hand, if requirement *R1* is modified to include a reference to a specific mode ("(M)" refers to "mode"), it will not conflict with requirement *R3*:

- *R1(M). In the* ***Idle mode****, the phone shall* ***initiate a call*** *when the PTT is pressed.*

Although both requirements imply the use of "transmit", none of them conflicts the other because the modes are orthogonal, i.e. there is no state of the phone when the two requirements are both relevant.

3.5. **Generating the Implied Actions List.** When searching for requirements which are based on actions that may conflict other requirements, DRAS

not only performs comparisons between actions directly used by the requirements, but also takes *implied* and *implying actions* into account.

3.5.1. *Implied and Implying Actions.* In many cases, the use of an action *Act* by a requirement implies the use of other actions by that requirement. Those are the actions which are the consequence of using *Act*. For example, the action "pressing the dial button on the phone" implies the use of the action ""transmitting voice". In addition, actions that imply the use of *Act* may also be relevant to the requirement. For example, when analyzing a requirement about "transmitting voice", the action "pressing the dial button" may also have to be considered.

We have to decide which actions to consider: those that are implied by the action *Act* or those that imply the use of *Act*. The decision depends on whether the requirement restricts the use of *Act* or whether it eases restrictions for the use of *Act*. Restricting the use of *Act* means that all actions that imply its use should also be restricted. Ease of restrictions for the use of *Act* means that all the actions that it implies should also be allowed.

Consider the case where an action (e.g., *transmit*) is restricted, i.e. it is not allowed in certain cases. The action that implies *transmit* (i.e., initiate a call) is also restricted:

- *R1. When the* ***PTT is pressed****, the phone shall* ***initiate a call****.*
- *R2. When* ***another phone transmits****, the phone shall not initiate voice* ***transmission****.*

In this case, since initiating a call requires the phone to transmit, a phone should not try to initiate a call if another phone is already transmitting. Note that this conclusion requires knowing that initiating a call results in a transmission.

3.5.2. *DRAS use of Implied and Implying Actions.* To identify the implied and implying actions for a certain action *Act*, the DRAS methodology uses predefined implied actions-list of all actions that are directly used by each action. That is, the actions-list specifies for each action A which other actions are implied by A. The implied actions-list is defined based on previous knowledge and during initial analysis of the system's requirements. Recursive use of the list allows it to identify all actions that are *implied* by the use of that action. For each implied action, the list also specifies whether it is always activated when *Act* is performed.

For example, for identifying that requirements *R1* and *R2* conflict, the list of actions implied by *call initiation*, as specified by the implied actions-list, is analyzed recursively to check if *transmit* is a result of *call initiation*. This

is shown in Figure 3 (*Tx* abbreviates transmit, see the Appendix for details about these actions).

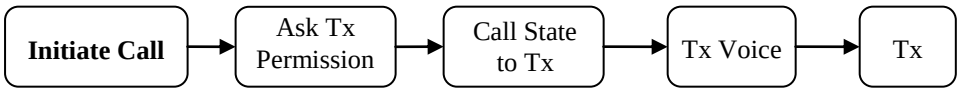

FIGURE 3. Implied Actions for Call Initiation

Given that *R2* has a higher or equal priority than *R1*, then the conflict resolution may be:

R1(E) When PTT is pressed, the phone shall initiate a call, unless another phone transmits.

A more complex example of implied actions is shown in Figure 4. It shows that several threads of actions can imply the same action; e.g., both *Power On* and *Initiate Call* imply *transmission*. Therefore, for example, not allowing transmission (Tx) means restricted functionality of call initiation and Power On.

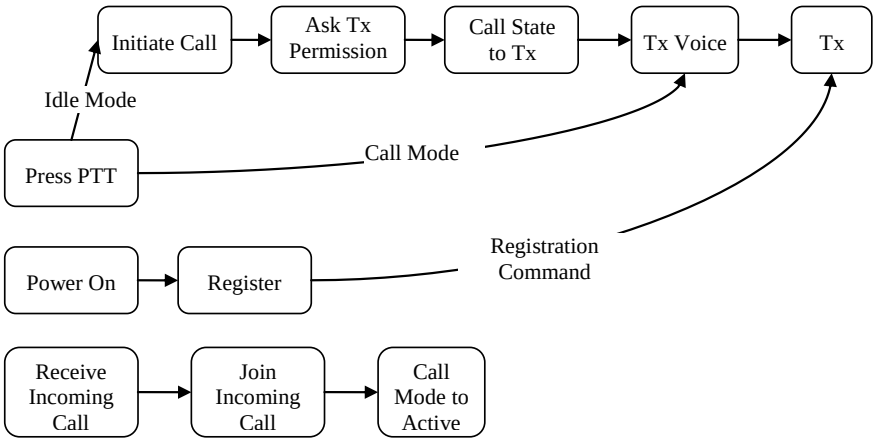

FIGURE 4. Implied Actions (a partial list)

To create the list of actions that are implied by other actions, the following steps are taken:

(1) The **list of actions** that are used by the requirements is identified.
(2) For each action, the **list of actions that they directly imply** (i.e., that are directly used by the action) are defined.

(3) The list of **all actions used by each action (either directly or indirectly)** is generated from the list of actions directly implied, as specified in the following pseudo code:

```
For each record in "Actions Implied by an Action" table{
    Create a record in "All Implied Actions" table with
    Action, Implied Action}
Repeat until no new record is added (no duplicates){
    For each two records in "All Implied Actions",
        where R1.Implied_Action = R2.Action{
           Create record in "All Implied Actions" table
           with R1.Action, R2.Implied_Action}}
```

3.6. **Identifying Implied Actions used by the Requirements.** In this step, the list of all actions used by the requirements, whether directly or indirectly (implied and implying), are identified, to ensure all actions common to the requirements are considered when checking whether a requirement conflicts another requirement. To identify whether implied or implying actions should be considered, the *action modifier* of the action in the requirement is used, as described below.

3.6.1. *Action Modifier.* Functional requirements usually conflict when they restrict normal functionality or ease other restrictions. For example, *R1* conflicts with *R2* since *R2* **restricts** the functionality of *R1*. On the other hand, *R4* **ease** the restriction of *R3* when the phone is in *Emergency* mode.

To allow identifying the type of possible conflict between requirements based on the actions they use, for each action used by a requirement, DRAS identifies its *action modifiers*. In general, an action modifier determines whether the requirement specifies when the action is used or when the action should not be used.

DRAS distinguishes between 3 action-modifiers:

- **Restrict**: an action is restricted or not allowed.
- **Unconditional**: an action is always allowed, even if it was restricted by other requirements (ease of restriction).
- **None**: an action is not specifically allowed or restricted in certain modes or states. Usually, an action with a non-action-modifier does not need to determine whether the FR conflicts other FRs or not.

The information regarding action-modifiers helps in determining whether two requirements are in conflict. If the use of an action is not restricted or a restriction for its use is not eased, then the use of the action does not necessarily mean the requirements are conflicting with other requirements. (An

exception is when there is a mistake in the requirements, such as: two contradicting requirements that are erroneously defined.) The action-modifiers are also propagated to the implied-actions.

It depends on the action modifier whether to consider the actions that are implied by an action or to consider the actions that imply the action. If a requirement **restricts** the use of action *Act*, all actions that **imply** *Act* are also restricted. For example, not allowing transmitting also means not allowing call-initiation, but not allowing call-initiation does not mean not allowing transmitting.

On the other hand, if a requirement **eases** the restrictions for using *Act* or allows using it unconditionally, then all actions **implied** by *Act* are also allowed. For example, permitting unconditional call-initiation in *Emergency* mode means also unconditional permission to transmit in this mode. Permitting unconditional transmission, however, does not mean unconditionally permitting call-initiation. The action-modifier of an action is therefore used to determine the *direction* for identifying implied-actions (see Figure 5). If *Act* is restricted, then the actions that imply *Act* are also restricted. If restrictions are eased (Unconditional), then restrictions for using the actions (implied by the action) are also eased.

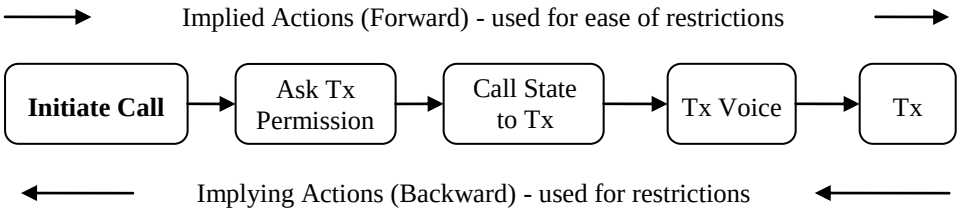

FIGURE 5. Restriction and Ease of Restriction for Implied Actions

For example, since *R2* restricts transmission, according to Figure 5, *R2* also restricts call-initiation; therefore, *R2* conflicts with *R1*.

Having only an action-modifier does not mean that one requirement may conflict with the other. Usually, in order to have a conflict the action-modifier should also contradict the action-modifier of the other requirement. For example, the following two requirements are not in conflict with each other:

R1 When the PTT is pressed, the phone shall ***initiate a call****. R4 In* ***the Emergency mode****, the phone should always be allowed to initiate a voice* ***transmission****.*

Although *R4* eases a restriction for transmission, it does not contradict *R1*, because *R1* refers to permitting transmission (implied by call initiation) and not to restricting transmission.

3.6.2. *Identifying Implied and Implying Actions.* Generating the list of actions used by a requirement is done as follows:

(1) The actions directly used by it are identified. For ach action, the *Action Modifier* is also identified, i.e. whether the action is *restricted*, it can be performed *unconditionally* (restriction is eased), or *none*.
(2) For each *restricted* action, the use of all actions implying it may also be *restricted*. The following pseudo code describes these actions list generation:

```
For each restricted Action1 used by the requirement{
    For each Action2 implying Action1{
        Add Action2 to the list of restricted actions
            used by the requirement}}
```

(3) For each *unconditional* action, the use of all actions implied by it may also be *unconditional*. The following pseudo code describes these actions list generation:

```
For each unconditional Action1 used by the requirement{
    For each Action2 implied by Action1{
        Add Action2 to the list of unconditional actions
            used by the requirement}}
```

(4) For each identified action, the *Action Modifier* of the implying or implied action and the requirement's *Priority* are assigned.

3.7. **Identifying Match-Points between the Requirements.** In this step, conflicting requirements and the requirements they are in conflict with are identified. This is performed by identifying the ***match-points*** between requirements, using their common attributes. The common attributes are *actions*, *modes*, *action modifiers* and other attributes that are common to the requirements. We also include *priorities* for the requirements.

For example, match-point will be identified between *R1* and *R2* as both use *voice transmission* (in *R1* voice transmission is implied by call initiation), and between *R1* and *R6* as they are both about *call initiation*.

Match-point identification is performed in three steps:

(1) Identify the list of match-point candidates between requirements, according to the use of common actions and different values for attributes. The following pseudo code describes this process:

```
For each pair of requirements using the same Action
(and per each common Action used by these requirements),
where the priority of the first requirement is
lower than the priority of the second requirement
and where the first requirement uses Action directly
```

```
and Action Modifier of both Actions is different {
    Add record to "Match by Action" table with Action
    and the list of attributes from both requirements}
```

(2) Remove redundancies that were created by multiple matches between two requirements (mainly caused by several implied actions that match between requirements). The following pseudo code describes this process:

```
Delete each record where a second record exists that
results from the same two requirements and with the
same Action
```

(3) Remove match-points that cannot happen in reality, or do not have at least one different conflicting mode or state.

```
Delete each record where two of its attributes contradict
or where the two requirements do not have at least one
different attribute
```

3.8. **Evaluating Match-Points.** After identifying the requirements that may conflict with each other, the effect of the conflicts should be evaluated. This is performed by identifying two attributes for each pair of matched requirements: *contribution* and *composition rules*, and is based on Brito and Moreira [7].

- ***Contribution*** - indicates how the function defined by one of the requirement affect the other requirement's function:
  - **Conflicts** with the functionality of the other requirement ("**-**")
  - **Adds** to that functionality ("+")
  - Does not affect the functionality ("**None**")
- ***Composition Rules*** - based on the requirements' relative priority and the nature of the conflicting functionality, the conflicting requirement can be one of the following:
  - **Overlap Before/After** - add functionality before/after the functionality of the requirement it conflicts with.
  - **Override** - replace the functionality
  - **Wrap** - encapsulate the existing functionality within new functionality.

Identifying these attributes enables an improved understanding of the nature of the conflict and how it can be resolved. After these attributes are identified, they are used for deciding how the requirements should be modified to resolve the conflicts, and to identify which of the *match-points* should result in a derived requirement.

For example, requirement *R2 conflicts* with *R1* and *override* its functionality, i.e. the phone should not initiate a call or transmit when other phone is transmitting:

- *R1. When the PTT is pressed, the phone shall* ***initiate a call****.*
- *R2. When* ***another phone transmits****, the phone shall not initiate voice* ***transmission****.*

On the other hand, *R4 overrides R2*, meaning that *R1* is allowed in emergency mode, even if other phone is transmitting.

R4 In the ***Emergency mode****, the phone should always be allowed to initiate a voice* ***transmission****.*

3.9. **Generating the Derived Requirements.** Derived requirements are generated according to the attributes defined for each *match-point*. In addition, requirements specifications may be enhanced based on the match-points attributes. Note that several *match-points* may result in one derived requirement, as several conflicts may have the same resolution.

For example, the following derived requirements may be generated (Rx/Ry indicates the matched requirements that are the source for the DRs):

- *R1/R2 When the PTT is pressed, the phone shall initiate a call, unless other phone is transmitting.*
- *R6/R7 Illegitimate users shall only be allowed to initiate a call to the police (an emergency number).*

4. Conclusion and Future Work

4.1. **Conclusion.** We have presented *DRAS*, a methodology to help identify and handle *conflicting functional requirements*. We have also applied our methodology to few test cases.

DRAS identifies conflicting functional requirements based on the actions they use. It starts with identifying the lists of actions and entities used by the input requirements. The relative priority of each requirement is also identified. Then the list of actions (implied by each action) is defined. This list is later used to identify all of the actions a requirement refers to, directly or indirectly. Generating the list depends on the action's action-modifier, i.e. whether the requirement restricts the use of an action or eases a restriction for its use. If the use of an action *Act* is restricted, the use of all actions that use *Act* (i.e. the implied-actions) is restricted too. If a requirement eases the restrictions for using an action *Act*, the actions list will include actions that are the result of using *Act* (i.e. the implying actions).

For each requirement, the modes and states of the different entities it refers to are also identified. This information is later used to help to decide whether

functional requirements conflict with each other, because this usually depends on the modes and the states referred to by the requirements.

The actions (and their modifiers) and modes for each requirement are identified. Based on this information, match-points between the requirements are identified. To get to the final list of match-points, the list is further refined to remove redundancies and conflicts that cannot occur in real-life.

The final step of DRAS is to generate DRs, according to the list of match-points between requirements. This process usually requires consulting the stakeholders; because in several cases resolving conflicts is not straight forward, and the stakeholders should decide what direction to take. The requirements, both original and derived, can be reviewed by all stakeholders, making sure that resolutions to conflicts are performed properly.

Using DRAS provides a method to identify conflicting functional requirements and the requirements they are in conflict with. It also helps in deciding what derived requirements should be generated from the conflict, by identifying the match-points between the requirements. By that, DRAS helps to define more consistent requirements.

4.2. **Further Work.** Several enhancements are considered for the DRAS methodology; mainly further automating the process beyond the partial automation of the prototype tool described in the Appendix, so that the (tentative) DRs can be generated automatically. Before automating the process, some more case studies should be done. In addition, automation requires the ability to parse and analyze the text and the ability to set the relative priorities between requirements a match-point refers. Note that text analysis should allow identifying actions, even when they are written in different forms. For example, "call initiation" may be written in the requirement "initiate a call", "start call", etc.

The DRAS methodology, or part of it, may be integrated with existing requirements management tools (such as DOORS or RequisitePro). This will enhance their functionality and enable an easier definition of requirements (derived from conflicts between other requirements). Another possible enhancement to such tools is the definition of attributes per requirement, as used in this work. For each requirement, these attributes include the actions used with their action-modifiers and the mode/state attributes. With proper textual analysis, the requirements management tool may be able to generate these attributes automatically. Using these attributes, the tool can suggest to the user possible conflicts between the requirements, by implementing similar algorithms to the ones defined for DRAS.

DRAS methodology assumes that an identified match-point (between functional requirements) tentatively identifies that one requirement is conflicting

the other. That is, one of the requirements is a conflicting requirement. This assumption was not validated; further work is required to identify whether this is true, or for what cases this is true.

Using natural language processing methods to analyze the requirements (e.g. [21, 32]), it may be possible to semi-automate actions identification of actions (used by the requirements) and their different attributes. Writing the requirements in some formal form, such as Attempto controlled language [35], can assist this approach. Ideas from AbstFinder [17] may also be used to help identify aspects in the specifications text. Mining aspects methods [23] and tools may also be used for automatic or semi-automatic retrieval and identification of aspects. Automatic weaving (composing) of requirements (to generate the DRs) may use methods similar to the ones used by aspect oriented programming (see [20]). As a starting point for using natural language processing, we consider adapting two tools that we have developed for other purposes. The EasyCRC tool [30], which automates the processes of finding nouns and synonyms, as part of its activity, can be adapted to find actions and related actions in the requirements. The CodePsychologist [27] is used to assist the programmer to locate regression bugs in the source code. It uses some affinity evaluation algorithms, which we consider using in our tool.

Using queries to identify conflicting requirements and the requirements they are in conflict with, as defined in the Requirements Description Language (RDL) [10], is another possible approach for enhancing DRAS. RDL identifies aspectual (conflicting) requirements by defining constraint queries about actions and objects used by the requirements. The requirements that the aspectual requirements tentatively are in conflict with are identified by base queries.

The use of XML to internally represent requirements can also be considered. Note that XML cannot be used to represent input and output requirements, because these should be in textual format, so that all stakeholders can understand. XML representation can help automate the creation of DRs. Methods will be needed to translate the textual requirements from text to XML (or another format) and to translate back the XML representation for DRs to textual format. XML is already used for aspect-oriented methods (e.g., the ARCaDe tool [31, 19]) to compose requirements, or for supporting aspects plug-ins in software design [22]. Concepts from these and other approaches may be reused.

To allow automatic detection of relative priorities between requirements, priorities may be added to each attribute value (e.g., Normal=1, Emergency=2). In addition to requirements priorities, this can also enable having relative priorities between requirements (i.e., a partially ordered tree of requirements

priorities). There will be no absolute priority per requirement, and the relative priority of each pair of requirements should be evaluated separately. In addition, default values for each attribute should be defined. This will enable requirements handling, where partial attribute values are specified (e.g., set call priority default as "Normal").

Composition-rules can be enhanced to improve the automation process. In many cases, current composition-rules values are not useful. Different values for composition rules, which are more suited for generating DRs, may be more useful. One possible approach is to define temporal rules, such as "Override Temporarily", "Delayed After", "On Event" (e.g., when mode changes). Enhancements using ideas from LOTOS [6, 8] should also be considered.

Temporal logic may also be used to enhance the method [24]. Action Modifiers identified in DRAS, "Restrict" and "Unconditionally" seem to be similar to Temporal Logic Path Quantifiers/Operators A/G (all paths / always) and A/H (all paths/always in the past). It may be possible to develop a logic based on temporal logic, that will use such action-modifiers and specify (using a formula), the effect of these aspectual action-modifiers on other requirements (e.g., Emergency $\rightarrow$ [A(always) PTT $\rightarrow$ Initiate Call]). The logic may be defined as an extension to already existing methods which support temporal logic for requirements, such as Formal Tropos [29] or Kaos [13]. Using formal languages that use temporal logic may allow the use of model checking methods [24] to identify conflicting requirements. The ideas suggested by [19] for the use of temporal logic in the PROBE framework can also be used as input for enhancements.

References

[1] J. Araújo, E. Baniassad, P. Clements, A. Moreira, A. Rashid, and B. Tekinerdogan. Early aspects: The current landscape. *Technical Notes, CMU/SEI and Lancaster University*, 2005.

[2] E. Baniassad and S. Clarke. Finding aspects in requirements with Theme/Doc. In B. Tekinerdoğan, P. Clements, A. Moreira, and J. Araújo, editors, *Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design*, pages 15–22, March 2004.

[3] E. L. A. Baniassad and S. Clarke. Theme: An approach for aspect-oriented analysis and design. In *ICSE*, pages 158–167. IEEE Computer Society, 2004.

[4] D. Bar-On and S. S. Tyszberowicz. Derived requirements generation: The DRAS methodology. In *SwSTE*, pages 116–126. IEEE Computer Society, 2007.

[5] D. Bar-On and S. S. Tyszberowicz. Aspects, dependencies and interactions: report on the workshop ADI at ECOOP 2007. In *Proceedings of the 2007 conference on Object-oriented technology*, ECOOP'07, pages 5–10, Berlin, Heidelberg, 2008. Springer-Verlag.

[6] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Comput. Netw. ISDN Syst.*, 14(1):25–59, March 1987.

[7] I. Brito and A. Moreira. Towards a composition process for aspect-oriented requirements. In J. Araújo, A. Rashid, B. Tekinerdogan, A. Moreira, and P. Clements, editors, *Early Aspects 2003: Aspect-Oriented Requirements Engineering and Architecture Design*, Mar. 2003.

[8] I. Brito and A. Moreira. Integrating the NFR framework in a RE mode. In *EA-AOSD: Workshop on Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design, held in conjunction with the 3rd International Conference on Aspect-Oriented Software Development, Lancaster, UK*, March 2004.

[9] I. Brito, A. Moreira, and J. Araújo. A requirements model for quality attributes. In *Workshop on Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design (AOSD-2002)*, March 2002.

[10] R. Chitchyan, A. Rashid, P. Rayson, and R. Waters. Semantics-based composition for aspect-oriented requirements engineering. In *Proceedings of the 6th international conference on Aspectoriented software development AOSD*, pages 36–48. ACM Press, 2007.

[11] R. Chitchyan, A. Rashid, P. Sawyer, A. Garcia, M. P. Alarcon, J. Bakker, B. Tekinerdoğan, S. Clarke, and A. Jackson. Survey of aspect-oriented analysis and design approaches. Technical Report D11 AOSD-Europe-ULANC-9, May 2005.

[12] L. Chung, B. Nixon, and E. Yu. *Non-Functional Requirements in Software Engineering*. Kluwer International Series in Software Engineering. Kluwer Academic, 2000.

[13] A. Dardenne, A. van Lamsweerde, and S. Fickas. Goal-directed requirements acquisition. In *Science of Computer Programming*, volume 20, pages 3–50. Elsevier, April 1993.

[14] G. M. C. de Sousa and J. Castro. Towards a goal-oriented requirements methodology based on the separation of concerns principle. In L. E. G. Martins and X. Franch, editors, *WER*, pages 223–239, 2003.

[15] A. Finkelstein and I. Sommerville. The view point FAQ. *Software Engineering Journal*, 1(11):2–4, 1996.

[16] M. Fowler and K. Scott. *UML distilled - a brief guide to the Standard Object Modeling Language (2nd edition)*. Addison-Wesley-Longman, 2000.

[17] L. Goldin and D. M. Berry. AbstFinder, a prototype natural language text abstraction finder for use in requirements elicitation. *Automated Software Engg.*, 4(4):375–412, October 1997.

[18] I. Jacobson. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2004.

[19] S. Katz and A. Rashid. From aspectual requirements to proof obligations for aspect-oriented systems. In *12th IEEE International Conference on Requirements Engineering (RE 2004), 6-10 September 2004, Kyoto, Japan*, pages 48–57. IEEE Computer Society, 2004.

[20] R. Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning Publications Co., Greenwich, CT, USA, 2003.

[21] D. Lin and P. Pantel. Discovery of inference rules from text. In *Seventh ACM SIGKDD international conference on Knowledge discovery and data mining, USA*, pages 323–328, August 2001.

[22] C. V. Lopes and T. C. Ngo. The aspect oriented markup language and its support of aspect plugins. Technical Report UCI-ISR-04-8, October 2004.

[23] N. Loughran and A. Rashid. Mining aspects. In *Workshop on Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design (AOSD-2002)*, March 2002.

[24] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. The Temporal Logic of Reactive and Concurrent Systems. Springer-Verlag, 1992.
[25] A. Moreira, J. Araújo, and I. Brito. Crosscutting quality attributes for requirements engineering. In *Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering*, pages 167–174. ACM Press, 2002.
[26] J. Mylopoulos, L. Chung, S. S. Liao, H. Wang, and E. S. K. Yu. Exploring alternatives during requirements analysis. *IEEE Software*, 18(1):92–96, 2001.
[27] D. Nir, S. S. Tyszberowicz, and A. Yehudai. Locating regression bugs. In K. Yorav, editor, *Haifa Verification Conference*, volume 4899 of *Lecture Notes in Computer Science*, pages 218–234. Springer, 2007.
[28] B. Nuseibeh. Crosscutting requirements. In G. C. Murphy and K. J. Lieberherr, editors, *AOSD*, pages 3–4. ACM, 2004.
[29] M. Pistore, A. Fuxman, Kazhamiakin, and M. R., Roveri. Formal Tropos: Language and semantics. Technical Report 4, November 2003.
[30] A. Raman and S. S. Tyszberowicz. The easycrc tool. In *ICSEA*, pages 25–31. IEEE Computer Society, 2007.
[31] A. Rashid, A. Moreira, and J. Araújo. Modularisation and composition of aspectual requirements. In M. Akşit, editor, *Proceedings 2nd International Conference on Aspect-Oriented Software Development (AOSD-2003)*, pages 11–20. ACM Press, March 2003.
[32] A. Sampaio, R. Chitchyan, and P. Rayson. Ea-miner: a tool for automating aspect-oriented requirements identification. In D. F. Redmiles, T. Ellman, and A. Zisman, editors, *ASE*, pages 352–355. ACM, 2005.
[33] A. Silva. Requirements, domain and specifications: a viewpoint-based approach to requirements engineering. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 94–104, New York, NY, USA, 2002. ACM.
[34] G. Sousa, G. Silva, and J. Castro. Adapting the NFR framework to aspect-oriented requirements engineering. In *Proceedings of the 17th Brazilian Symposium on Software Engineering (SBES)*, pages 177–192, 2003.
[35] H. Stefan. The syntax of Attempto controlled english: An abstract grammar for ace 4.0. Technical Report ifi-2004.03, 2004.
[36] Y. Yu, J. Leite, and J. Mylopoulos. From goals to aspects: Discovering aspects from requirements goal models. In *12th IEEE International Requirements Engineering Conference*, September 2004.
[37] ETSI EN 300 392-2. Terrestrial Trunked Radio (TETRA); Voice plus Data (V+D); Part 2: Air Interface (AI), 2007.

Motorola Solutions Israel
E-mail address: David.Bar-On@motorolasolutions.com

School of Computer Science, The Academic College of Tel Aviv Yaffo
E-mail address: tyshbe@tau.ac.il