Finding Aspects In Requirements with Theme/Doc

Elisa Baniassad and Siobhán Clarke
Department of Computer Science
Trinity College, Dublin 2, Ireland
{Elisa.Baniassad, Siobhan.Clarke}@cs.tcd.ie

Abstract

Aspects are behaviours that are tangled and scattered across a system. In requirements documentation, aspects manifest themselves as descriptions of behaviours that are intertwined and interdependent. Some aspects may be obvious, as specifications of typical crosscutting behaviour. Others may be more subtle, making them hard to identify. In either case, it is difficult to analyse requirements to locate all points in the system where the aspects should be applied. To identify aspects early in the software lifecycle developers need support for aspect identification and analysis in requirements documentation. To address this, we have devised the Theme/Doc approach for viewing the relationships between behaviours in a requirements document to identify and isolate aspects in the requirements. This paper describes the approach, and illustrates it with a case study and analysis.

1. Introduction

Conceptually, an aspect is an element of functionality that is woven throughout other system behaviours. At the source level, aspects are tangled and scattered code. At the requirements level, aspects are tangled and scattered descriptions of functionality. In a requirements document, tangled functionality can only be described in relation to other functionality, whereas scattered functionality is described throughout the requirements document.

Tangling and scattering at the requirements level makes it difficult for developers to reason about whether they have encountered aspects. It is difficult to mentally note whether a behaviour that is scattered across a document is actually dependent on (tangled with) other behaviours, or just badly encapsulated in the requirements set. Additionally, it is often difficult for a developer to keep in mind which behaviours are tangled.

These difficulties place a barrier between a developer and full adoption of aspect-orientation, because they make it difficult for developers to conceive of whether they have "crosscutting behaviours" in their set of requirements.

Using intuition or even domain knowledge is not necessarily sufficient for identifying the potentially broad range of aspects within a reasonable amount of time. For instance, developers might start by looking in their documentation for typical aspect-style behaviour, such as logging, tracing, or debugging functionality, or non-functional requirements, but this likely does not cover the full range of potential aspects.

We assert that developers need support for viewing and manipulating their requirements to expose how elements of functionality relate to one another.

To address this need, we propose the Theme/Doc approach, which provides views of requirements specification text, exposing the relationship between behaviours in a system. These views assist a developer in determining which elements of functionality are "aspects" and which are "base". Theme/Doc views also provide a feature-oriented view of a requirements set. All views can be mapped to Theme/UML models. Theme/Doc and Theme/UML together comprise the *theme approach*.

1.1. Theme/Doc

Theme/Doc is based on the notion of a *theme*, which represents a feature of a system. Multiple themes can be combined to form a functioning whole according to a multi-dimensional model [10]. There are two kinds of themes: *base themes*, which may share some structure and behaviour with other base themes, while modelling these from their own perspective, and *crosscutting themes* which have behaviour that overlays the functionality of the base themes. Crosscutting themes are *aspects* [3].

The Theme/Doc tool operates on the basic assumption that if two behaviours are described in the same requirement,[1] then they are related. Behaviours can relate in three

---

[1] We currently take a requirement as being a sentence in a requirements document, or a single requirement in a set of requirements. Because of the lexical nature of the tool, however, the granularity and format of a

ways: they can be erroneously or coincidentally related, meaning that the requirement could be re-written so that they were no longer coupled, they can be hierarchically related, in that one behaviour is a sub-behaviour of another, or they can be related by crosscutting, meaning that the requirement describes one behaviour as an aspect of another. The Theme/Doc tool provides views that expose which behaviours are co-located in requirements. These views assist the developer in determining what kind of relationships exist between behaviours, and whether those behaviours are base or aspects.

In Section 2 we outline how to apply the Theme/Doc approach and tool. We then present a case study (Section 3) in which we apply the approach on a larger example. Next we raise issues for discussion (Secion 4), and review related work (Section 5). Finally, we conclude (Section 6).

2. A Small Example

In this section we work through the example of a small Expression evaluation system to illustrate the basic points of how to use Theme/Doc to support the identification of aspects in a set of requirements.

2.1. Expression System Requirements

1. evaluation capability which determines the result of evaluating expression.

2. display capability which depicts expression textually.

3. check-syntax capability which determines whether expression are syntactically correct.

4. log capability that logs the evaluation display and check-syntax activities.

5. an expression is grammar-defined as a variableexpression or a numberexpression or a plusoperator or a minusoperator or a unaryplusop or a unaryminusop.

6. a plusoperator is grammar-defined as an expression and a plus and an expression.

7. a minusoperator is grammar-defined as an expression and a minus and an expression.

8. a unaryplusop is grammar-defined as a plus and an expression.

9. a unaryminusop is grammar-defined as a minus and an expression.

10. a variableexpression is grammar-defined as a letter and an expression.

11. a numberexpression is grammar-defined as a number and an expression.

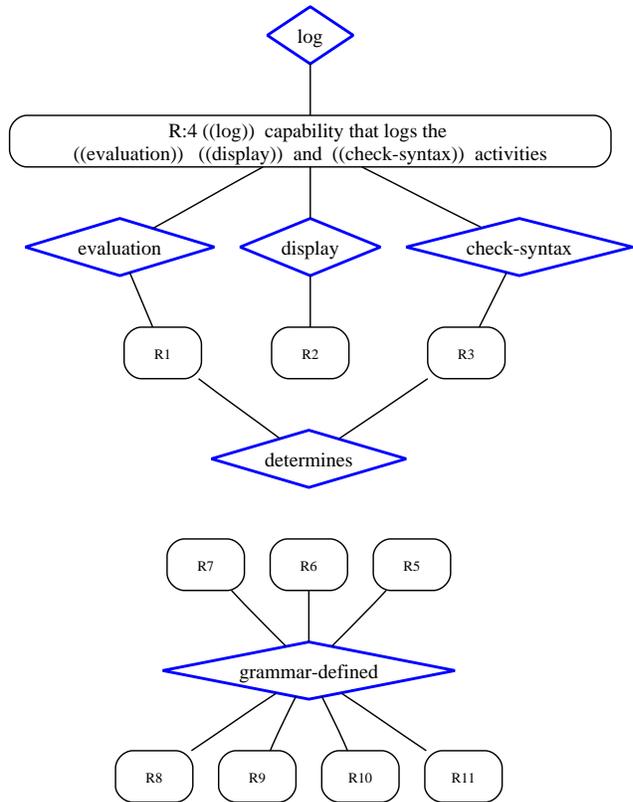requirement can be set arbitrarily.

Figure 1. Action View

2.2. Identify Actions and Entities

The tool takes the requirements, as written above, as input. Behaviours in requirements are identified by a set of keywords provided by the developer to the Theme/Doc tool. These behaviours are referred to as "actions". Strictly speaking, any lexical string, including a non-functional requirement, state, etc. can be made a "key-action" if it was considered a candidate theme. However, we have found that using actions is a good starting point for finding themes, and that requirements that seem not to contain actions can often be refined to include actions. The developer also provides a set of key-entities as input. Entities can also encompass resources. It uses these inputs to generate the Theme/Doc views.

For this set of requirements, we identify six actions: evaluation, display, determine, check-syntax (the whole concept, not just the verb "check"), log, and grammar-defined. We also identify nine entities: expression, variable-expression, number-expression, plus-operator, minus-operator, unary-plus-operator, unary-minus-operator, plus and minus.

2.3. Categorize Actions into Themes

Theme allows the individual design of different system features. In Object-orientation, not all the nouns in a requirements document are designed as objects or classes. Similarly, in Theme, not all actions are designed as separate features of the system: some actions are sub-behaviours of other actions. In this step, we set out to identify the features, or themes of the system, by identifying which actions are major-enough to be modeled separately (once we get to the point of modeling). For this, we use a Theme/Doc view called the *action view*.

Figure 1 shows the action view for the Expression system. An action view consists of two elements: actions, shown as diamonds, and requirements, shown as rounded boxes. If an action is mentioned in a requirement, there is a line linking the action to the requirement.

In this view, we can either see requirements as labels (their requirement number), or we can enlarge them to see their content, as we have for R4. The action view is non-hierarchical, so even though it looks as though some actions are "higher" than others, this is just a coincidence of layout.

We use this view to determine whether actions should be themes, or just behaviour (perhaps methods) within themes. Deciding which actions are not major enough to be a theme is a highly intuitive process. We scan these actions and question whether it makes sense to have each of them as a feature in our system. If they are not feature-worthy, we demote them from our action view. The remaining "major actions" will be our themes. The "log" feature makes sense as a theme: it is something that we would, perhaps, like to turn on or off, or at least model separately from the other actions we see in the view. A "check-syntax" feature makes sense for the same reason, as does a "display" feature, an "evaluation" feature, and a feature centrally responsible for defining the grammar (the "grammar-defined" theme). The "determines" action, however, does not seem to be as strong a potential theme as the rest. It is involved in two requirements, but in a relatively minor way and it seems hard to imagine it as a collection of classes. It is more likely a sub-behaviour: a method, rather than a feature in and of itself. For this reason, we decide to demote "determines" from our action view.

2.4. Identify Crosscutting Themes

We use the *major action view* to help us determine which themes are base, and which are aspects. This view is made up only of the major actions from the previous action view. It is identical to the view shown in Figure 1 except the "determines" node is removed.

Our focus in using this view is on the requirements that are shared by more than one theme. If a requirement is
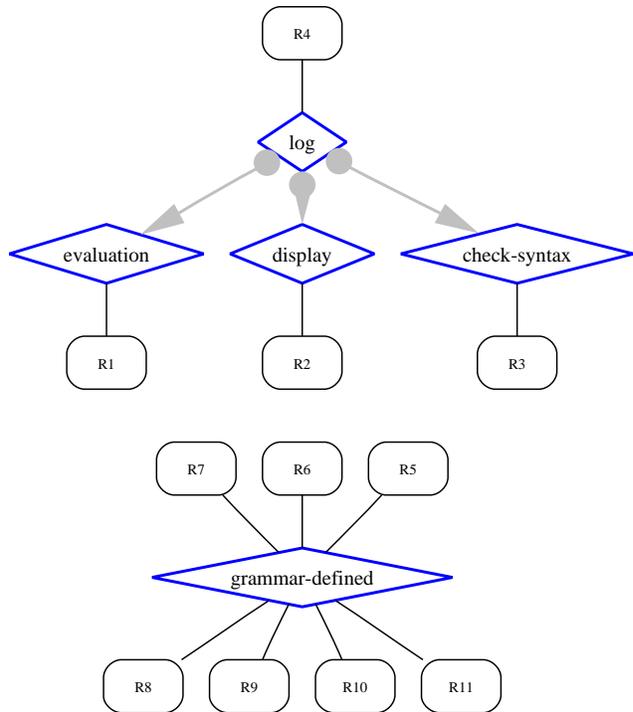

Figure 2. Clipped Action View

shared by two or more themes, we must decide which theme should provide that functionality. Shared requirements flag for us that we may have identified an aspect in our system, since they imply that two themes cannot operate without behaviourally relying on one another.

As we can see, some requirements are already associated with only one theme. It is straightforward to assume that whatever functionality should be associated with those themes. So, when designing the check-syntax theme we know it should implement the functionality described in R3.

We begin our investigation by inspecting R4 (shown in Figure 1) which is shared between several themes. We want to ensure that we have not encountered a vaguely written requirement that is masquerading as a shared one. So, we check to see whether the requirement can be re-written into several requirements that each refers to only one theme. However, we can see that in this case the only possible re-writing would be to break it into three sentences that each mention log, and another of the themes (a log capability that logs the evaluation activity; a log capability that logs the display capability, etc). There is no re-writing that gets a 1-1 relationship between themes and requirements: the log feature must be overlaid on the evaluation, display, and check-syntax features. This means that the log theme is an aspect.

We denote that the log theme crosscuts the other three associating the shared requirement, R4, with it. This association clips the links from R4 to the other three themes. In its place a grey arrow indicating a crosscutting relationship is placed from the aspect theme to the base themes (Figure 2). It is the developer's job to ensure that this 1-1 relationship is achieved. If a particular shared requirement is too ambiguous to make the association clear, then it is up to the developer to revisit the requirements set, and resolve the ambiguity.

There are no other shared requirements, so we can now move on to examining our themes individually, and planning for design. The product of this process is referred to as the *clipped action view*. The grey arrows in this view indicate the crosscutting-hierarchy.

2.5. View Individual Themes

Themes (at this point a "major action" is the same as a theme) can be viewed individually as well as grouped in the action views. An individual theme view shows the requirements associated with the major action, as well as minor actions mentioned in the requirements. Key entities are also shown individually in this view, as boxes. These views are used to check that the associations were made correctly when manipulating the major action view to form the clipped view. They can also be used to determine how themes should be modeled using Theme/UML [1].

3. Case Study

The goals of this case study were to test the Theme approach on a larger example, and perform preliminary assessment of it in terms of effectiveness for finding aspects, support for assessment of requirements coverage, and scalability. We will first give a general description of the location aware game that was the basis for the case study, and then provide results and analysis.

3.1. Location Aware Game

The set of requirements used in this case study are those for a location aware game called the Crystal Game, which was developed in an independent research project. The game has 89 requirements, so is roughly eight times larger than the example provided in Section 2. The object of the game is to collect crystals that are deposited throughout the location. As a player moves around the game space, their hand-held device will alert them when they have encountered a crystal. Computer-generated characters also take part in the game. When a player encounters one of them, they will interact and perhaps duel. When a player encounters another player, they will duel, and the loser will turn all
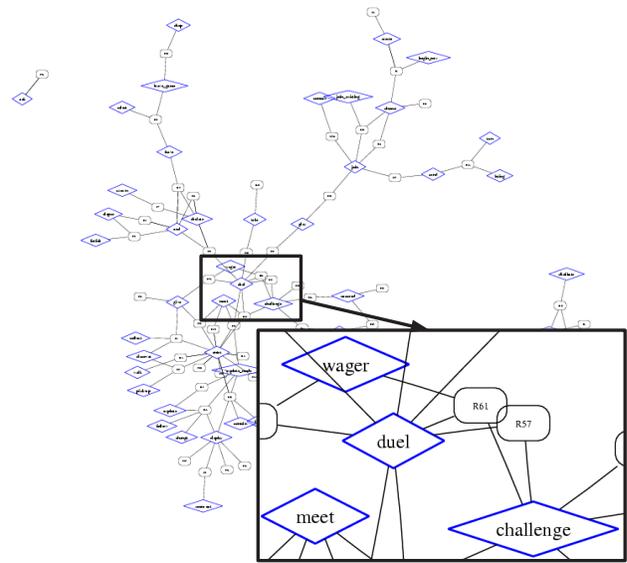

Figure 3. Game Action View: All Actions

of their crystals over to the winner. The game ends after a specified time period. The winner is decided by how many crystals each player has. There are other constraints and requirements in this game which will be of interest and will be described in later sections.

3.2. Results

In this section we review the steps we took to apply the Theme approach to the Crystal Game requirements.

3.2.1. Finding Themes

We identified 59 actions in the game requirements, and generated an action view to examine their relationships. Based on intuition and some cursory analysis of the view, we determined that all of these actions should not be modeled as separate themes. Instead, we examined the view to determine the relationships between the actions, to decide how to group the actions into larger themes.

This was a mainly analytical process, but it was supported by the action view. Because actions that share requirements are displayed close to one another in the view, we were able to examine closely located actions to assess whether they should be grouped into a common theme.

We used the view shown in Figure 3 to perform such an assessment. This figure shows the initial action view for the game, with the centre portion of the view enlarged. The enlarged view shows four actions, *duel, wager, challenge* and *meet*. We examined the requirements they shared, considered the meaning of the actions, and determined that *duel,*
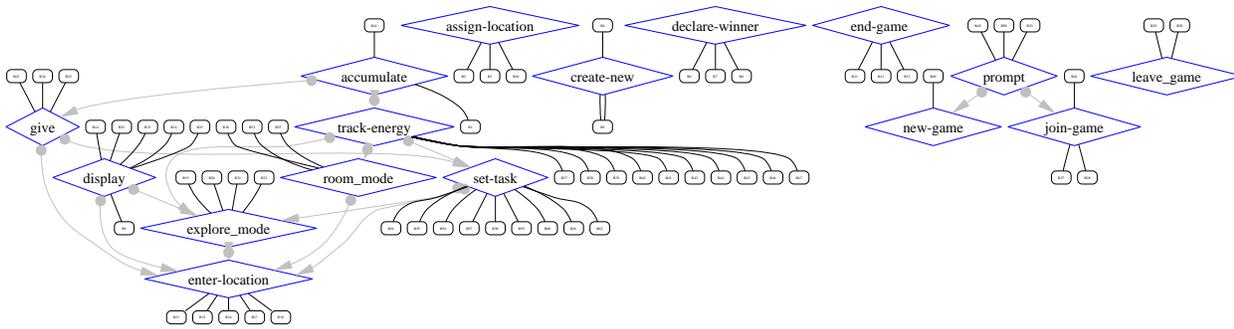
Figure 4. Clipped Action View of Major Game Actions

wager, and *challenge* should all be grouped under the general heading of *duel*, since players challenge one another to duel, and wager crystals on the outcome of a duel. In that case, we classified *duel* as being more major than *wager* and *challenge*, which we saw as sub-actions of *duel*. The *meet* action was connected to *duel* because when players encounter one another they duel. We examined requirements shared by *meeting* and *duelling* and determined that since they were not synonymous, they should not be grouped into one theme. Later, we determined that *duel* and its sub-actions should be grouped under the more major action, *set-task*. In the end, we arrived at the view shown in Figure 4, which displays the 16 major actions which became our themes. Of those actions, five are independent, while others share requirements, and hence crosscut one another in some way.

The clipping functionality of the tool helped us investigate the major action view to determine which themes are crosscutting and which are base. In the case of the *prompt* theme this was straightforward. The *prompt* theme (shown to the right of Figure 4) shared requirements with two other themes, *new-game* and *join-game*. By examining the shared requirements it could be seen that the prompting behaviour crosscut these two themes.

As is visible on the left side of Figure 4, there are several related themes. To determine which of those was crosscutting, we began by assessing the requirements between the *explore-mode* and *enter-location* themes. We determined that *explore-mode* crosscut *enter-location*. By continuing to examine themes that shared requirements with *enter-location* we further determined that *room-mode* was crosscutting, as was *give, accumulate, set-task,* and *display*. We then examined the remaining shared requirements, and encountered themes that crosscut other crosscutting themes. For instance, the *track-energy* theme was determined to crosscut *set-task, room-mode* and *explore-mode*, all of which crosscut *enter-location*. There are five themes that crosscut other crosscutting themes: *display, set-task, give, track-energy* and *accumulate*.

3.2.2. Determining Composition Order of Themes

We used the crosscutting relationships shown in the clipped action view (Figure 4) to determine the order of binding. In this view, the themes are positioned hierarchically, based on whether they crosscut one another. The grey arrows indicate which themes crosscut other themes. We can see, for instance, that there are no grey arrows extending from *enter-location*, which indicates that it is base functionality. To determine the binding order, we begin with the lowest themes in the crosscutting-hierarchy, and work to the highest, incrementally binding in one crosscutting theme at a time. To determine what is first in the binding we identify the themes that crosscut only that theme (*room-mode* and *explore-mode*), and placed those first, and second in the binding order. The final bindings were done with the "more" crosscutting themes: *display, give* and *track-energy*. The very last binding is of *accumulate*, since it crosscuts the *give* and *track-energy* themes. It is not intended that establishing the aspect-base associations at the Theme/Doc level guarantees that conflict-free theme composition at the modeling stage. Theme/Doc is only intended to be used to assist visualization of the relationships between elements of functionality described in the requirements as opposed to a design/model checker.

3.3. Analysis

In this section we discuss how the results of the application of the Theme approach reflect on its effectiveness at support for aspect identification, requirements coverage, and on its scalability.

3.3.1. Effectiveness of Support for Aspect Identification

Through the application of the Theme approach, we were able to identify eight aspects: *explore-mode, room-mode, accumulate, track-energy, give, set-task, display* and *prompt*. Had we carefully read the requirements document

we may have identified seven of these behaviours as aspect behaviours since they provide tracking or logging style functionality. However, it is unlikely that we would have identified the *give* functionality as an aspect because mentions of the *give* action are spread throughout the document, and it might have been difficult to recall that the same abstract behaviour is occurring with relation to different system features. Also, since in the document text it is described as a consequence of other actions, such as meeting, and duelling, it is possible that we would have automatically thought of *give* as a method in those actions. It wouldn't have been until we were modelling or implementing it that we would have noticed its crosscutting nature.

We also found our approach effective support for determining the binding order for multiple crosscutting themes. This may be otherwise difficult to determine.

3.3.2. Requirements Coverage

We were initially concerned that it may be difficult to assess whether all the requirements have been associated with a theme. We noted that the action view can be used to monitor requirements coverage, because if a requirement is not associated with a theme it is orphaned in the view. By orphaned, we mean that only the sentence record for the requirement appears, without being linked to a diamond-shaped action. We found that a requirement could be orphaned in two ways. Orphans can appear in the initial action view if the requirement contains no key actions. This may happen if it refers to another action, but does not mention it explicitly. By inspecting the requirement we can identify the requirement's original location in the text, and can read the requirement in context to determine to which action it refers. The other way orphans can appear is when forming the major action view. As major actions are identified, they are added to a new list of keywords. The minor actions that have been grouped under the major action will be annotated so that they will be linked to the major action in the view. Minor actions that are not grouped with major actions will disapear, and their requirements will appear to be orphaned. We systematically visited the orphaned requirements to determine whether any of their minor actions should be promoted to major, or whether to group those requirements under other major actions.

3.3.3. Scalability of Action Views

When applying Theme/Doc, actions are classified into two types: major and minor. Major actions become themes, while minor actions were slotted to become methods within a theme. This approach has essentially provided two "zoom-levels" of action view: a developer can zoom-in to see all the actions, or can zoom-out and just see the major actions. This approach worked very well with the small

Expression example, and was very useful for the Crystal Game.

However, were we to scale the requirements further, it would be necessary to apply other approaches, since it would not be feasible to fit an entire major action view for a very large system onto a screen or a page. In this case, query functionality is needed to form sub-views that could be examined separately from the entire action view. Additionally, it would likely be useful, in a larger system, to provide more degrees of zooming so at some level the entirety of the system could be seen in one view.

4. Discussion

In this section, we provide discussion of issues we noted while performing our case study.

4.1. Synonyms

Synonyms are handled through a synonym dictionary which, for the sake of the action view, automatically augments the requirements text so that the correct associations will be made. This is more complicated when two words are the same but have different meanings in terms of the system. For instance, the term *give* was used in the Crystal Game not only for giving crystals, but also for giving audio and visual signals to players. The action view helped identify instances where this occured, because the common action brought together other actions which, upon analysis, should not be linked. For instance, the common term *give* brought closer together *accumulate* and *prompt*. We could intuit from having read the requirements document that these two actions should be unrelated. When inspecting the relationships around the *give* action, it was clear that the term was being used in different senses. We then used the annotation feature of the tool to replace the audio sense with the term *give-audio*. These annotations are not shown in the theme view.

4.2. Ambiguities Found in Requirements

We found that the Theme approach helped us identify ambiguities in the original requirements specification. While refining the 59 actions into the 16 themes we found that there were subtle ambiguities in the initial requirements document. For instance, we found that it was not explicitly mentioned how crystals were collected by a player, unless the crystal was actually given to them by another player or a game character. One requirement mentioned picking up crystals ("a player explores the world and picks up crystals"), and another mentioned the accumulation of crystals ("a player collects crystals by discovery in a location, or when a player or character gives one to them.") Though it

was implied, there was no specific description of a player actually picking up a crystal when they discover it in a location. This subtle ambiguity was discovered when we saw that the *pick-up* action, and the *collect* action were not located close to one another in the view shown in Figure 3, though we knew intuitively that they should be related. This was highlighted because the *collect* action was closely placed to the *give* action.

4.3. Evolution of Requirements

Neither our example nor our case study considered what would happen were the requirements to change over time. There are two situations in which requirements can change: during the requirements gathering and analysis stage, or after modeling has begun. In the former situation, the evolution can be handled by re-generating the views for the set of requirements, and deciding which themes the new requirements should fall under. A developer would follow the same process as outlined for the original requirements set for the new or changed set of requirements. This is also a possibility in the latter situation, after modeling has begun.

5. Related Work

There have been several efforts in capturing and relating aspect-oriented requirements [9, 11, 4, 8, 7, 6, 2]. Here we consider the two which relate most closely to the Theme approach.

Rashid et al [8] provide the AORE (Aspect-Oriented Requirements Engineering) model and ARCaDe (Aspectual Requirements Composition and Decision support) approach and tool for describing components and requirements-level aspects. Examples of these aspects are compatibility, availability, or security. This work builds on the ViewPoints model [5], which is intended to support the integration of heterogeneous requirements specified from multiple perspectives. An early stage in the AORE model is the identification and specification of concerns. The approach to this differs from the Theme approach to concern identification in that it relies on the domain knowledge of the developer to identify possible non-functional requirements to be taken into account when implementing a particular requirement. Those concerns are not explicitly mentioned in the requirements specification; it is up to the developer to ascertain their relevance on their own. We see this as a complemetary approach to our own. Such domain knowledge will always play a large part in system design. The Theme/Doc approach aims to support the analysis of relationships between behaviours described in requirements specifications. It is possible that the Theme/Doc approach to aspect identification could be used during the concern identification phase

of AORE, or could support AORE's extension to include functional as well as non-functional requirements.

Katera and Katz [7] propose architectural views of aspects as a means for reasoning about the relationships among aspects in a system. They describe aspects as cross-cutting augmentations to an existing design. In particular, they allow for specification of the overlap between aspects through the concept of a *sub-aspect* that provides the overlapping functionality, and they make relationships between aspects explicit. A UML approach is given to support these views which differs from the Theme/UML approach: it provides additional architectural support for aspect modelling to that provided by Theme/UML, and it uses aspect mappings rather than multi-dimensional composition style semantics. Theme/Doc could be integrated into this approach since the relationships exposed between behaviours in a set of requirements could be used to establish the behaviours between aspects and sub-aspects in this approach, as well as support the identification of functionality shared between components.

6. Conclusions

In order to identify aspects in a set of requirements, we need to see how behaviours described in the requirements relate to one another. In this paper we have presented Theme/Doc, which provides views of requirements specifications that are intended to expose relationships between behaviours in requirements. Our case study showed that this approach is effective in helping to identify aspects in requirements, and helped us identify functionality that would enhance the scalability of the approach.

7. Acknowledgements

References

[1] E. Baniassad and S. Clarke. Theme: An approach for aspect-oriented analysis and design. In *International Conference on Software Engineering (to appear)*, 2004.

[2] J. Castro, M. Kolp, and J. Mylopoulos. Towards requirements-driven information systems engineering: The tropos project, 2002.

[3] S. Clarke and R. J. Walker. Composition patterns: An approach to designing reusable aspects. In *International Conference on Software Engineering*, pages 5–14, 2001.

[4] R. Darimont and A. van Lamsweerde. Formal refinement patterns for goal-driven requirements elaboration. In *Foundations of Software Engineering*, pages 179–190, 1996.

[5] A. Finkelstein. The viewpoints faq. *BCS/IEE Software Engineering Journal*, 11(1), 1996.

[6] J. Grundy. Aspect-oriented requirements engineering for component based software systems. In *4th IEEE International Symposium on Requirements Engineering*, pages 84–91.

[7] M. Katera and S. Katz. Architectural views of aspects. In *Proceedings of the International Conference on Aspect-oriented Software Development*, pages 1–10, 2003.

[8] A. Rashid, A. Moreira, and J. Araujo. Modularisation and composition of aspectual requirements. In *Proceedings of the International Conference on Aspect-oriented Software Development*, pages 11–20, 2003.

[9] S. Sutton. Early stage concern modeling. In *Early Aspects Workshop, Held with AOSD*, 2002.

[10] P. Tarr, H. Ossher, W. H. Harrison, and S. S. Jr. N degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of the International Conference on Software Engineering*, pages 107–119. IEEE Computer Society Press, 1999.

[11] X. Wang and Y. Lesperance. Agent-oriented requirements engineering using congolog and i*. In *Submitted to AOIS-2001, Bi-Conference Workshop at Agents 2001 and CAiSE'01.*, 2001.