Analysis of Crosscutting in Model Transformations

Klaas van den Berg\(^1\), Bedir Tekinerdogan\(^1\), Hoa Nguyen\(^1\)

\(^1\) Software Engineering Group, University of Twente
7500 AE Enschede, the Netherlands
\{K.G.van.den.Berg, B.Tekinerdogan, H.Nguyen\}@utwente.nl
http://www.trese.cs.utwente.nl

Abstract. This paper describes an approach for the analysis of crosscutting in model transformations in the Model Driven Architecture (MDA). Software architectures should be amenable to changes in user requirements and technological platforms. Impact analysis of changes can be based on traceability of architectural design elements. Design elements have dependencies with other software artifacts but also evolve in time. Crosscutting dependencies may have a strong influence on modifiability of software architectures. We define crosscutting based on a traceability pattern. We present an impact analysis of crosscutting dependencies in transformations rules at model level as well as at metamodel level.

1 Introduction

Change management is a prerequisite for high-quality software development. Changes may be caused by changing user requirements and business goals or be induced by changes in technologies and platforms. Software architectures must be designed such that they can evolve to cope with these changes. The Model Driven Architecture (MDA) approach aims at providing stable models amenable to changes [24]. An analysis of the impact of changes is necessary for cost effective software development [3]. The number of affected modules or elements is a response measure for the quality attribute modifiability in software architectural design [7]. Such analysis can be based on dependency traces between elements in the architectural design and other software artifacts. We will elaborate on traceability research in requirements engineering [20][28]. Especially in case of crosscutting dependencies, the impact can be large. Crosscutting has been studied extensively in the context of Aspect-Oriented Software Development (AOSD) [17].

Traceability is defined as the degree to which a relationship can be established between two or more products of the development process, especially products having a predecessor-successor or master-subordinate relationship to one another [21]. In [20], traceability is defined in the context of requirements engineering, and a distinction is made between pre-requirements specification traceability (forward to requirements and backward from requirements) and post-requirements specification traceability (forward from requirements and backward to requirements). Moreover, one may distinguish inter-level trace dependencies (sometimes called horizontal traceability)
and intra-level trace dependencies (or vertical traceability) [1]. In Figure 1, several types of traceability are shown. For example, architectural design elements are traced backwards to requirements and forward to elements in the detailed design. In MDA, similar links can be defined in model transformations of CIM, PIM and PSM [24]. Elements at the architectural design may have intra-level dependency relations, and may evolve to a new configuration of architectural elements. In [28], a meta-model for requirements traceability is discussed, together with instances of traceability links. There are traceability links between artifacts (such as requirements and architectural design elements) and links representing the evolution and/or incremental development of these artifacts.

![Figure 1. Traceability in Software Development](image)

Traceability is an optional requirement in the Query, Views and Transformations (QVT) Request for Proposal (RFP) issued by OMG [26]. In response to the RFP, eight proposals were submitted to OMG in October 2002. In [19], these proposals have been evaluated. During the year 2003 and 2004, the submissions were revised and converged. In March 2005, two remaining submissions were combined to provide the joint 3rd revised submission by the QVT Merge Group [27]. This QVT specification use trace classes for keeping traces between source and target elements which are mapped in a transformation. In the Relations Language, these trace classes are not specified explicitly. A rule in this language directly specifies the relationship between source and target domain elements. On the other hand, in the Core Language, a trace class is specified explicitly for each transformation mapping. This QVT specification also proposes transformation rules to transform a transformation definition in the Relations Language to the corresponding one in the Core Language, in
which there is a rule to define the a trace class for each relation. The trace class contains a property corresponding to each object node in the pattern of each domain of the relation [27].

In this paper, we present a framework for change impact analysis in case of crosscutting in model transformations (see Figure 2). The impact analysis is based on traceability of dependencies between elements in software artifacts. We propose a graph representation of dependencies, together with formal definitions of specific cases of dependencies, such as tangling, scattering and crosscutting. We define crosscutting dependencies between elements and show that crosscutting may have an impact on rule execution in case of changes.

![Figure 2. Relation between topics discussed in this paper](image)

The paper is structured as follows. In Section 2, we give our definition of crosscutting based on a traceability pattern. We describe how to represent crosscutting in dependency graphs and present a change impact analysis of crosscutting. In Section 3, we apply these definitions in the context of model transformations using a case study. Finally in Section 5, we describe related work and present conclusions of the paper.

### 2 Crosscutting and Change Impact Analysis

The problem of crosscutting has been studied in AOSD. Crosscutting can occur at implementation level, but also in early phases of software development [17]. In previous papers [8][9], we generalized the concept of crosscutting by means of a crosscutting pattern (see Figure 3). We call this a pattern as in [18] because it is a recurring problem (obvious from Figure 1) for which we propose a conceptual framework for solutions. In this pattern, we have dependency relations between elements in the source and elements in the target. These dependency relations also provide traceability between the elements, as well as inter-level relations as intra-level relations. Intra-level relations denote coupling between elements at a certain level. There is extensive literature on different types of coupling and the trade-off between coupling and cohesion (e.g. [14]). Here, we focus on inter-level dependencies.
Our proposition is that crosscutting can only be defined in terms of 'one thing' with respect to 'another thing'. In other words, at least two domains (or two levels or two phases) are related with each other in some way.

- A level could refer for example to models in the four-level metamodeling architecture (e.g. M0, M1, M2, ..) [30].
- A domain could refer for example to models with a certain refinement in the Model Driven Architecture (e.g. CIM, PIM and PSM) [24].
- A phase could refer to any phase in the software development lifecycle (e.g. requirements analysis, design, implementation, and so on).

We use here the general terms source and target (as in [24]) to denote two consecutive domains, phases or levels. We assume that elements in the source are related to elements in the target: there is a mapping between source and target elements. The mapping can be established manually or be automated in transformation rules. In the context of MDA, these transformation rules can be defined in terms of the source metamodel and the target metamodel as in the metamodel transformation pattern [24].

**Figure 3. Traceability Pattern for Crosscutting**

We may distinguish several cases of mappings between source and target. This can be represented in a dependency graph, as shown in Figure 4

- Injection: a source element is related to a distinct target element (e.g. s2 to t2)
- Scattering: a source element is related to multiple target elements (e.g. s1 to t1, t3 and t4)
- Tangling: a target element is related to multiple source elements (e.g. s1 and s3 to t3)
- Crosscutting: a target element is involved both in scattering and tangling (e.g. t3; scattering of s1 to t1, t3 and t4, and tangling of s1 and s3 in t3)

We say that source element s1 crosscuts source element s3 with respect to the given mapping between source and target.

According to our definition in [8], crosscutting occurs when in a mapping between source and target, a source element is mapped to two or more target elements and at least one of these target elements has a mapping from one other source element. Al-
though crosscutting is defined as a relation between source elements (see Figure 3), the crosscutting relation depends on the mapping between source and target. Coupling is also defined as a relation between elements at source level; however this relation just depends on intra-level relationships.

![Diagram of mapping between elements at different levels of abstraction](source.png)

**Figure 4. Mapping between elements at different levels of abstraction**

In broad sense, we may define crosscutting just as scattering and tangling [17]. We defined crosscutting - in narrow sense - as a specific combination of tangling and scattering. We only considered dependencies between two levels. Usually we encounter multiple levels as shown in Figure 1. In that case we have to take into account the transitivity of trace dependency relations. This can be accomplished through the cascading of the dependency pattern as described in [10].

We consider change impact in case of tangling, scattering and crosscutting, both for two-level dependencies and multiple-level dependencies. Change impact in the traceability pattern is operationalized in terms of the elements involved in the change of a source element. The set of elements is called the impact set. We now show some examples of the change impact in case of a change in some source element for different cases of mappings between sources and targets (Figure 5).

<table>
<thead>
<tr>
<th>Source Type</th>
<th>Impact</th>
<th>Change Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Injection</td>
<td>Impact s1</td>
<td>change (s1,t1)</td>
</tr>
<tr>
<td>Scattering</td>
<td>Impact s2</td>
<td>change ((s2,t2), (s2,t3))</td>
</tr>
<tr>
<td>Tangling</td>
<td>Impact s3</td>
<td>change (s3,t4) + preserve (s4,t4)</td>
</tr>
<tr>
<td>Crosscutting</td>
<td>Impact s5</td>
<td>change ((s5,t5), (s5,t6)) + preserve ((s6,t6), (s7,t6))</td>
</tr>
</tbody>
</table>

Changing s1 (for injection) means that t1 need to be changed. Similarly, changing s2 (for scattering) means that t2 and t3 need to be changed. Changing s3 (for tangling) means that t4 needs to be changed, however in this change, the dependency of s4 in t4 has to be preserved. Changing s5 for crosscutting means that t5 and t6 need to be changed, while preserving the dependency of s6 in t6, and of s7 onto t6.

The impact set consists of two subsets: impacted target elements that need to be changed (forward traceability from source to target), and impacted source elements that need to be preserved in this change of target elements (backward traceability from target to source).

The severity of anticipated impact for injection is relatively weak, for scattering and tangling the anticipated impact is moderate, whereas for crosscutting the antici-
pated impact is strong. The severity of the actual impact in each case depends on the number of elements involved and eventually on the type of change required.

Figure 5. Examples of change impact

Above, we only considered dependencies between two levels. Usually we encounter multiple levels as shown in Figure 1. In that case we have to take into account the transitivity of trace dependency relations. Assume we change element v in the mapping (x,v), then the mappings involved in this change can be obtained from the following recursive function:

\[
\text{impact}(x,v) = \\
\text{change}(x,v) + \left( \text{preserve}(u,v) \mid u \leftarrow \text{preds}(v); x \neq u \right) + \\
\text{impact}(v,w) \mid w \leftarrow \text{succs}(v)
\]

where the function \text{preds} gives the predecessors of an element (adjacent elements in backward trace) and the function \text{succs} gives the successors of an element (adjacent elements in forward trace). The result of this function contains a list of mappings to be changed based on forward traceability, and a list of mappings to be preserved based on backward traceability. In [11], we elaborate this approach.

3 Case Study: Concurrent File Versioning System (CFVS)

As example, we consider a simple Concurrent File Versioning System (CFVS) [5]. A versioning system allows the user to keep multiple versions of data ordered by a timestamp (see UML class diagram in Figure 6). A user may get an existing version of data from the system, modify that data and save it back to the system as a new version without discarding the previous version of that data. Each version may either be a complete copy of the modified data or a log of changes to the previous version of data.

A basic file versioning system has the following basic functionalities: check-in, check-out, commit, update, remove and difference. The commit functionality allows the user to create a new version a file in the system with the local copy in the user workspace, while the update functionality does the opposite: to update the local copy in the user workspace with the content of a specific version of the file in the system. The update functionality requires conflict management; i.e. when the local copy of the file in the user workspace and the current version of the file in the system are
modified simultaneously. The check-in and check-out functionalities, besides the similar operations as commit and update, respectively, manage the locking mechanism. The check-out functionality locks the file before retrieving the content, whereas the check-in functionality creates a new version of the file in the system and releases the lock on that file. The locking mechanism is necessary for concurrent access to a file in the system by multiple users. The remove functionality deletes the specified version of a file from the system. The difference functionality shows the differences between two versions of a file.

![Class Diagram in Design Model of CFVS](image)

A user may get an existing version of data from the system, modify that data and save it back to the system as a new version without discarding the previous version of that data. Each version may either be a complete copy of the modified data or a log of changes to the previous version of data.

A basic file versioning system has the following basic functionalities: check-in, check-out, commit, update, remove and difference. The commit functionality allows the user to create a new version a file in the system with the local copy in the user workspace, while the update functionality does the opposite: to update the local copy in the user workspace with the content of a specific version of the file in the system. The update functionality requires conflict management; i.e. when the local copy of the file in the user workspace and the current version of the file in the system are modified simultaneously. The check-in and check-out functionalities, besides the similar operations as commit and update, respectively, manage the locking mechanism. The check-out functionality locks the file before retrieving the content, whereas the check-in functionality creates a new version of the file in the system and releases the lock on that file. The locking mechanism is necessary for concurrent access to a file in the system by multiple users. The remove functionality deletes the specified
version of a file from the system. The difference functionality shows the differences between two versions of a file.

The basic file versioning system is extended for software configuration management (SCM). Essential for SCM is the tracking and maintaining the integrity of many different files. For this requirement, branching and tagging are needed. Tagging is the ability of the versioning system to mark a specific version of each of these files to belong to a specific version of the software product. Branching is the ability to branch a set of tagged files off the main trunk or another branch. A branch can only be conducted after the set of files has been tagged. Merging is the ability to merge a branch back to the original branch from which it was originated.

![Sequence Diagram in Design Model of CFVS](image)

The CFVS is concurrently used by multiple users with different permissions. This means that it has to implement the security functions to manage the permissions of each user. The two most fundamental security functions are authentication and authorization. Authentication is the requirement that the user must be identified before using the system. Each user has a username and a password which are entered at the beginning of his or her working session with the system. The system checks whether the username exists and the password matches. Authorization is the ability of the system to check whether a user is allowed to do some action on a particular object.

Based on this analysis, we devised a structural model of the CFVS (UML class diagram) in Figure 6 and a behavioral model (UML sequence diagram) in Figure 7.

3.1 Model transformations between UML design model and Java code model

In this section, we restrict our analysis to the transformation of this design model (UML class diagram and sequence diagram) to a Java model, based respectively on a
(subset of) the UML metamodel and (a subset of) a Java metamodel. Only two transformation rules are discussed. The first transformation rule UClassToJClass (TR1) establishes the relation between each class in the UML model with a unique java class in the Java model (see Table 1).

<table>
<thead>
<tr>
<th>top relation UClassToJClass</th>
</tr>
</thead>
<tbody>
<tr>
<td>ucn: String;</td>
</tr>
<tr>
<td>checkonly domain uml uc:Class (name=ucn);</td>
</tr>
<tr>
<td>enforce domain java jc:JavaClass (name=ucn);</td>
</tr>
<tr>
<td>where {</td>
</tr>
<tr>
<td>AttributeToField (uc, jc);</td>
</tr>
<tr>
<td>OperationToMethod (uc, jc);</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>class TFromClassToJClass {</td>
</tr>
<tr>
<td>uc: Class;</td>
</tr>
<tr>
<td>jc: JavaClass;</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>

Table 1: TR1: UClassToJClass transformation rule and its QVT tracing class

The tracing model for this transformation definition includes a QVT class for this rule which has two members of the types Class (imported from the UML metamodel) and JavaClass (imported from the Java metamodel), respectively. When executed, the rule locates all classes in the UML model and, for each class, determines whether there exists a Java class with the same name in the Java model; if none exists, then the Java class is created. An instance of the tracing class is also created in the transformation engine for this pair of UML and Java classes.

<table>
<thead>
<tr>
<th>top Relation MessageToImports</th>
</tr>
</thead>
<tbody>
<tr>
<td>checkonly domain uml msg:Message {</td>
</tr>
<tr>
<td>sendEvent = sendMsgEnd:MessageEnd {</td>
</tr>
<tr>
<td>covered = sendLife:Lifeline {</td>
</tr>
<tr>
<td>type = sendClass:Class {}</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>},</td>
</tr>
<tr>
<td>receiveEvent = recvMsgEnd:MessageEnd {</td>
</tr>
<tr>
<td>covered = recvLife:Lifeline {</td>
</tr>
<tr>
<td>type = recvClass:Class {}</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>enforce domain java importingJClass:JavaClass {</td>
</tr>
<tr>
<td>importedClass = importedJClass:JavaClass {}</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>when {</td>
</tr>
<tr>
<td>UmlClassToJavaClass(sendClass, importingJClass);</td>
</tr>
<tr>
<td>UmlClassToJavaClass(recvClass, importedJClass);</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>class TMessageToImports {</td>
</tr>
<tr>
<td>msg: Message;</td>
</tr>
<tr>
<td>sendMsgEnd: MessageEnd;</td>
</tr>
<tr>
<td>recvMsgEnd: MessageEnd;</td>
</tr>
<tr>
<td>sendLife: Lifeline;</td>
</tr>
<tr>
<td>recvLife: Lifeline;</td>
</tr>
<tr>
<td>sendClass: Class;</td>
</tr>
</tbody>
</table>
Table 2: TR2: MessageToImports transformation rule and tracing class

The purpose of the second transformation rule MessageToImports (TR2) is to transform call messages modeled in the sequence diagram to corresponding elements in the Java model (Table 2).

A complete transformation is to create appropriate statements in the Java method corresponding to the UML operation that is described by the interaction diagram. Here as illustration, the transformation rule only creates the imports relation between two Java classes corresponding to the UML classes participating in the call message. The tracing class for this transformation rule includes members for the Message element, the UML Class elements participating in the Message element, and the corresponding JavaClass elements.

3.2 Analysis of crosscutting based on trace dependencies at metamodel level

In this section, we analyze the transformation rules TR1 and TR2 in terms of dependencies at metamodel level. For TR1, it is straightforward to derive the dependency graph between source and target elements as shown in Figure 8.

![Figure 8: Dependency graph of TR1 at metamodel level](image)

The example rules TR1 and TR2 are given in the QVT Relational Language. The QVT Core Language groups variables of the rules into different areas and patterns as explained in the QVT Specification [27]. The dependency graph of the second rule TR2 is now explained in terms of the Core Language (see Table 3).

It is observed from this specification that only the target element importingClass depends on the source elements msg, sendMsgEnd, recvMsgEnd, sendLife, recvLife, sendClass and recvClass. In addition, there is a difference between the mapping relations from source elements to target elements in terms of change effect. With respect to this rule, any change to msg, sendMsgEnd, recvMsgEnd, sendLife or recvLife will cause immediate changes to importingJClass, or the change is direct. On the other hand, when there is a change with either sendClass or recvClass, this change may be propagated to sendLife, recvLife, sendMsgEnd, recvMsgEnd and msg, and consequently to the target element importingClass; this kind of change is
considered indirect. These mappings are called direct and indirect relations, respectively.

<table>
<thead>
<tr>
<th>Source area</th>
<th>Middle area</th>
<th>Target area</th>
</tr>
</thead>
<tbody>
<tr>
<td>sendClass: Class</td>
<td></td>
<td>importedJClass: JavaClass</td>
</tr>
<tr>
<td>recvClass: Class</td>
<td></td>
<td>importingJClass: JavaClass</td>
</tr>
<tr>
<td>msg: Message</td>
<td></td>
<td>Guard patterns</td>
</tr>
<tr>
<td>sendMsgEnd: MessageEnd</td>
<td></td>
<td></td>
</tr>
<tr>
<td>recvMsgEnd: MessageEnd</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sendLife: Lifeline</td>
<td></td>
<td></td>
</tr>
<tr>
<td>recvLife: Lifeline</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3: Allocation of variables in rule TR2

The determination of whether a relation is direct or indirect is also based on the patterns as shown in Table 3. The relations from the source elements in the bottom pattern of the source area to the target elements in the bottom pattern of the target area are direct, while the relations from the source elements in the guard pattern of the source area are indirect.

Figure 9: Dependency graph of TR2 at metamodel level

The dependency graph (Figure 9) between source and target elements are derived based on the definition of the rule in the Core Language in which the direct and indirect mapping relations are created from each element in the bottom pattern and each element in the guard pattern of the source are to each element in the bottom pattern of the target area, respectively. In terms of tangling, scattering and crosscutting, in this rule we only encounter tangling. This implies that in case of a change in a source element, we have to change the mapping, but also preserve the other mappings to the target element (i.e. importingJClass).
3.3 Analysis of crosscutting based on trace dependencies at model level

In this section, we analyze the transformation rules TR1 and TR2 in terms of dependencies at model level. The dependency graph for the first rule is shown in Figure 10. One or more numbers are attached to each mapping relation to mark the rule number and the number of the tracing data record representing a specific application of this rule; this number is called the rule application number.

In the UML design model, operations are modeled in sequence diagrams to show the usage of the Security classes by the Repository classes (Figure 7).

![Figure 10: Dependency graph of TR1 at model level (selection)]

In the UML design model, operations are modeled in sequence diagrams to show the usage of the Security classes by the Repository classes (Figure 7). Based on the mapping derivation method above, the transformation execution of rule TR2 can be represented in the dependency graph as shown in Figure 11.

![Figure 11: Dependency graph for TR2 at model level (selection)]

(P = Product; B = Branch; S = Security)

The dependency graph of TR2 at model level illustrates cases of tangling, scattering and crosscutting within a single rule. We can use or change impact analysis of crosscutting to see whether or not rules need to be re-executed in case of changes in some model element. For example, assume that there is a single change to the source element "P.add P-S Msg", the rule application 2.1 needs to be re-executed. However,
when the source element Product changes, besides the re-execution of the rule application 1.1, the rule application 2.1 is re-executed only when this change also makes changes to the source element “P.add P-S Msg”.

The combined dependency graph should be used to identify the order of re-execution of these rule applications. Figure 12 is the combination of the two dependency graphs introduced above.

![Combined dependency graphs of TR1 and TR2 at model level](image)

Figure 12: Combined dependency graphs of TR1 and TR2 at model level

The element Product is involved in crosscutting which implies the need for re-execution multiple rules in case of changes. When the source element “P.add P-S Msg” is modified, the rule application 2.1 needs to be re-executed. However, this rule application interacts with other rule applications because it contains indirect relations from source elements Product and Security. If the source element Security is changed as well, then the rule application 1.5 needs to be re-executed first. In general, it is possible to order the execution of rule applications based on this combined dependency graph as follows: any of the rule applications in the group (1.1, 1.2, 1.3) should be executed before any of the rule applications in the group (2.1, 2.2, 2.3); rule applications in the same group can be executed in any order.

In this case study, it is shown that crosscutting dependencies may occur at model level and at metamodel level. Changes in model and/or metamodel elements will have impact on elements that depend on the changed elements as can be seen from the dependency graphs. A special case is encountered when a mapping to a target element has to be changed and at the same time another mapping to the same target element has to be preserved. These cases require special consideration in the ordering of rule execution and/or incremental compilation. More details about the case study are provided in [25].
4 Related Work

In this paper, we used graphs to represent dependencies between elements at different levels. These graphs can be represented in dependency matrices (see also [8]). Several authors use matrices (design structure matrices, DSM) to analyze modularity in software design [6]. The design structure matrices represent intra-level dependencies (as coupling matrices) and do not address the inter-level dependencies as in the dependency matrices used for our analysis of crosscutting. In project management, an extension to design structure matrices is proposed in [13]. In so-called domain mapping matrices (DMM), they capture the dynamics of product development. In their terminology, the traditional DSMs support intra-domain analysis, whereas the DMMs support inter-domain analysis. The purpose of these design mapping matrices is similar to our inter-level dependency graphs as presented in this paper.

Traceability is a build-in feature of some model transformation language (ATL [23] and Mistral [22]). However, no change impact analysis is based on traceability data. In [4], a number of events and change actions have been defined as part of an operational semantics of traceability. These events and change actions could be the start of a change impact analysis as described in this paper.

In [29], the generation of traceability links is discussed, especially between requirements and the object model, and between requirements. This corresponds to the inter-level dependencies and the intra-level dependencies as described in this paper. Similarly in [15], traceability links are retrieved between UML and target models including one-to-many relations. In the current paper, we focused on generated trace relations as part of QVT transformations. An event-based approach to traceability is described in [12]. In this approach, change is handled by means of event notification and propagation of changes using traces between artifacts.

5 Conclusion

In this paper, we described an approach for the analysis of crosscutting in model transformations. We defined a traceability pattern and defined specific cases of trace dependencies represented in dependency graphs. We analyzed the change impact in case of tangling, scattering and crosscutting. Two types of impacted mappings are distinguished: mappings that need to be changed and mappings that need to be preserved (while changing source and/or target).

The specification of transformation rules and the tracing information of the execution of these rules can be used to generate the individual and combined dependency graphs at model level and at metamodel level. These dependency graphs are helpful in identifying which rule applications need to be re-executed in which order when there are changes to the source model. These are the key issues to implement incremental model compilation of transformation languages.

Our approach needs further elaboration and has to be validated in empirical case studies. Moreover, the derivation of dependencies and its analysis should be supported by tools in order to scale to industrial projects.
Acknowledgement

This work has been carried out in conjunction with the AOSD-Europe Project IST-2-004349-NoE [2] and of the ESI Project Darwin [15].

References