Removing Energy Code Smells with Reengineering Services

Marion Gottschalk, Mirco Josefiok, Jan Jelschen, Andreas Winter

Department of Computer Science
Carl von Ossietzky University Oldenburg
Ammerländer Heerstr. 114-118
26129 Oldenburg
{gottschalk, josefiok, jelschen, winter}@se.uni-oldenburg.de

Abstract: Due to the increasing consumer adoption of mobile devices, like smart phones and tablet PCs, saving energy is becoming more and more important. Users desire more functionality and longer battery cycles. While modern mobile computing devices offer hardware optimized for low energy consumption, applications often do not make proper use of energy-saving capabilities. This paper proposes detecting and removing energy-wasteful code using software reengineering services, like code analysis and restructuring, to optimize the energy consumption of mobile devices.

1 Introduction

The increasing energy consumption of information and communication technology is creating a rising demand for more energy-efficiency (cf. [SNP+09]). It is important to reduce the energy consumption of mobile devices to preserve environmental resources and maintain an acceptable level of energy consumption caused by information and communication technology. Also, users of devices want to be independent of current power sources, but battery technology develops slower than the devices’ functionality [Wue11].

Many opportunities exist for reducing energy consumption on different levels, ranging from hardware, operating system, machine code to application level [JGJ+12]. Various research focuses on low-level software optimization; e.g. in improving machine code [RJ97]. Another approach is to optimize hardware components for reducing energy consumption of mobile devices (cf. [HB11, Kam11]). In software engineering, it is best practice to find and remove errors (in this case: energy wasteful code) as early as possible for optimizing energy consumption of applications on every level. The work presented in this paper, focuses on possibilities for improving energy-efficiency on application level by applying reengineering techniques to applications.

Viewing energy-efficiency on application level requires analyzing and interrogating code structures. Improving energy consumption of applications necessitates changing and reworking source code. Altering source code for improving software qualities is, viewed as perfective maintenance, targeting energy consumption. In the field of software evolution, various techniques have been developed during the last decades, which have been successfully applied to improve software systems. This paper aims at applying these techniques for lowering energy consumption of applications by finding energy wasting patterns in the
application’s source code. These patterns will be called energy code smells, analogously to code smell detection in software maintenance. Software maintenance views code smells as source code segments which have to be restructured for improving software quality including maintainability [FBB+02].

Developing energy-efficient software is discussed in various papers. Höpfners work classifies software components by utilizing software complexity in O-notation [HB10]. They also developed a component-based and model-driven framework, which estimates and optimizes the energy consumption of a software system. A vision for self-aware systems and services was given by Kounve [Kou11]. They combined different areas e.g. software and system engineering, cloud computing and Green IT, to reduce the costs of information and communication technology. Next to analyzing hardware aspects, Pathak’s research focuses on applications of mobile devices to detect energy code smells (here called energy bugs) by tracing system calls [PJHM11]. Energy bugs are an equivalent to energy code smells, but address a wider range. They describe not only energy wasteful code patterns on application level, but also on operation system and hardware level of smartphones. Moreover, Pathak et al. [PJHM11] aim to develop a systematic diagnosing framework for debugging energy bugs. In contrast, the focus of this work lays on applying reengineering techniques for removing energy code smells. Additional ideas for energy savings on application level are presented in a vision paper, which focuses on a model-based energy testing approach [WGRA11]. In it they try to predict energy consumption by using a combination of abstract interpretation and run-time profiling. Another idea for reducing energy consumption on software level is described by Siegmund [SRA10]. There, an energy-optimization feature library for storing reusable energy-saving functionalities is created. This library could be used by developers without knowledge about energy-saving algorithms.

The remainder of this paper is organized as follows: Foundations of reengineering and reverse engineering services and their applications for improving energy-efficiency are presented in Section 2. An example refactoring and its capabilities for removing energy code smells is given in Section 3. Section 4 adds further types of energy code smells, their detection techniques and possible restructurings. Section 5 concludes this paper.

2 Software Reengineering

Software reengineering defines the process of altering software with the purpose of adding functionality or correcting errors [CC90]. Figure 1 shows a basic reengineering reference framework [EKRW02] introducing the main reengineering steps. Source code of applications is parsed and stored into a central repository which provides efficient code analysis. In software evolution, these repositories are usually found on graph structures which conform to a metamodel (cf. e.g. [ERW08]). All analyses and restructurings, required during reengineering, are performed on graphs. Explicitly defined metamodels ensure a clear, precise and targeted definition and documentation of underlying data structures [JCD02].

Applying reverse engineering techniques, source code is transferred into higher abstractions, like control flow or call graphs. Further reverse engineering provides querying software systems, e.g. to expose interrelationships between various software concepts. In
addition to static analysis, dynamic analyses are possible. Dynamic analyses are used to observe users activities and to learn from it, e.g. to shut down an application after a specific time and hence to reduce energy consumption \cite{CZvD09} depending on users behavior. Static analysis influences the restructuring process, marked by 3, which aims at converting existing code to more energy-efficient code. After restructuring, the graph is unparsed to source code and played back to the systems source code base 4.

The reengineering reference framework in Figure 1 is split into two areas: (i) The repository and facilities to parse, store, and unpars source code according to a specific metamodel is described in Section 2.1. (ii) Section 2.2 introduces reverse-engineering and restructuring techniques to provide energy aware refactorings. Refactoring combines energy code smell detection via reverse engineering, and restructuring of source code to improve energy-efficiency.

2.1 Repository

Transferring source code of a given application into reasonable data structures is required for performing advanced analyses efficiently. For optimizing the energy consumption of applications for mobile devices, analyses must address the source code level. In this work TGraphs \cite{ERW08} are used to represent source code. TGraphs are directed graphs, whose nodes and edges are typed, attributed, and ordered. TGraph-based tooling is provided for supporting reverse engineering activities. All necessary functionalities for applying TGraphs are embedded in JGraLab \cite{Kah06}. The structure of TGraphs is defined by UML class diagrams forming appropriate and purposeful metamodels. Classes define node types and edge types are specified by associations.

Figure 2 shows an extract of a complete Java metamodel, which contains all concepts needed to explain the examples in Section 3. This excerpt was extracted from the SOAMIG Java metamodel \cite{FWE12} intending to support migrating Java programs. The complete
SOAMIG Java metamodel contains 86 node types and 67 edge types and provides a fine grained representation of Java code, which is also required for analyzing energy-efficiency on code level. The metamodel is accompanied by a parser translating Java in an appropriate TGraph. Also generating source code from a given TGraph, is realized with the SOAMIG unparsertools developed by pro et con, Chemnitz [FWE+12].

Java programs are represented by Class-nodes and methods are represented by Method- Type-nodes. To provide unambiguous links to source code, DataObjects associated to various nodes, are used to store fully qualified names. Therefore, DataObjects connect Classes to declared (by HasMethod-edges) and called methods (by CallsMethod-edges). The SOAMIG parser also lifts method calls to class level, such that all methods (DataObjects) are connected by HasMethod-edges to calling Classes.

Figure 2: Java Metamodel (extract) [FWE+12]

2.2 Software Evolution Services

Software evolution requires various activities and techniques to be applied to software systems. These activities are viewed as services which allow for a flexible handling and usage of services [JW11]. Refactoring is a widely applied reengineering service, which aims at improving the internal structure of programs without altering its external behavior [FBB02]. According to [ISO06] refactoring is viewed as service providing perfective maintenance, i.e. improving software quality without modifying the system’s behavior. Improving software systems energy consumption by detecting and resolving energy code smells also intends to keep the software behavior apparent to the user.

Applying refactorings for energy-efficiency provides detecting programming faults effecting in dissipation of energy and restructuring the code, accordingly. Fowler defined more than 70 refactorings in his book sorted in different categories [FBB02]. In this paper a
similar list of refactorings on basis of energy-efficiency is intended. Energy code smells were described by a motivation and a technique for detecting and restructuring them.

Already applying Fowler’s standard refactorings might influence energy consumption of applications (cf. [Sho09]). Applying e.g. the in-line method refactoring which exchanges a method call for its body [dSB10] reduces energy consumption by avoiding to create additional activation records. On the contrary, this refactoring will probably reduce maintainability, since code clones might be created.

The remainder aims at presenting refactorings directly oriented towards energy-efficient software development. To accomplish these refactorings on graphs, code smell detection is mapped to graph queries using GReQL (Graph Repository Query Language). GReQL is a declarative expression language for analyzing TGraphs, which can be applied to various reverse engineering services e.g. calculating cross references, software metrics, program slicing [KW99]. Figure 5 in Section 3 gives an example of a GReQL query, used to detect binding resources too early.

3 Refactoring for Energy-Efficiency

To demonstrate an energy code smell and its detection and restructuring, an Android example is presented in this section. Whilst the general approach in this work is generic, this section provides an example using Android as a concrete platform. For different platforms the mechanisms are the same, but the structure of the restructuring process has to be adapted. The life cycle of an Android application is shown in Figure 3, represented as a state machine based on the Android documentation [Goo12].
Android applications pass through various states during their lifetime. Considering these states and their associated activities, allows for analyzing and optimizing the energy consumption of applications. When an application is started, `onCreate()` is called when entering `ApplicationLifetime`. Leaving `ApplicationLifetime` results in calling `onDestroy()`. User directly interact with applications in `Foreground`. Entering this state effects in calling `onResume()` and leaving calls `onPause()`. Applications in `Background` still perform relevant calculations, but probably do not request all resources required for user interaction. If neither the user interacts with an application nor the application is running in background, but still is present e.g. for faster restart, it is in `Invisible`. Usually, applications do not require access to further resources in this state.

If applications are in `Background` or `Invisible`, usually addressed resources like GPS sensors, Wi-Fi etc., are not requested. When being up, they consume energy and switching them on or deactivating them will save energy. Applications wasting energy may switch on resources too early when starting the application in `onCreate()` and do not switch them off when sent to sleep. A more economic behavior will be caused, if resources are only switched on if needed. This behavior can be achieved, if these resources are switched on only if being in `Foreground`.

**Figure 4: Excerpt of Android App GPS Print.**

```
public class GpsPrint extends Activity
    implements OnClickListerner, Listener , LocationListener {

    [...]

    public void onCreate(Bundle savedInstanceState) {
        [...]
    }

    [...]

    LocationManager lm=(LocationManager) this.getSystemService(Context.LOCATION_SERVICE);
    if (lm.getAllProviders().contains(LocationManager.GPS_PROVIDER)) {
        if (lm.isProviderEnabled(LocationManager.GPS_PROVIDER)) {
            lm.requestLocationUpdates(LocationManager.GPS_PROVIDER, 1000, 0, this);
            status_view.setText("GPS service started");
        }
        else {
            status_view.setText("Please enable GPS");
            save_location_button.setEnabled(false);
            [...]
        }
    [...]

    public void onPause() {
        [...]
    }

    [...]

    public void onResume() {
        [...]
    }

    [...]

    public void onDestroy() {
        [...]
    }
}
```

Before Refactoring

```
public class GpsPrint extends Activity
    implements OnClickListerner, Listener , LocationListener {

    [...]

    public void onCreate(Bundle savedInstanceState) {
        [...]
    }

    [...]

    LocationManager lm=(LocationManager) this.getSystemService(Context.LOCATION_SERVICE);
    if (lm.getAllProviders().contains(LocationManager.GPS_PROVIDER)) {
        if (lm.isProviderEnabled(LocationManager.GPS_PROVIDER)) {
            lm.requestLocationUpdates(LocationManager.GPS_PROVIDER, 1000, 0, this);
            //removed by refactoring
            status_view.setText("GPS service started");
        }
        else {
            status_view.setText("Please enable GPS");
            save_location_button.setEnabled(false);
        }
    [...]

    public void onPause() {
        [...]
    }

    [...]

    public void onResume() {
        [...]
    }

    [...]
}
```

After Refactoring
Figure 4 shows part of the implementation of an open source Android application GPS Print (Version 0.5.2) [Rob12], which displays geographic coordinates of the user’s current position on the screen. The code on the left shows the original code before refactoring. Starting GPS Print, immediately activates the GPS, by calling requestLocationUpdates() (line 16). When setting up, the applications also verifies, if the GPS sensor is enabled; if not the user is asked to enable it (line 22). If the user accesses GPS Print, i.e. the application moves to Foreground, the GPS is initialized in line 33 by calling requestLocationUpdates(), again. The GPS is released in onPause() by calling removeUpdates() (line 29).

Following the state machine in Figure 3, initializing GPS Print, results in sequentially accessing the states Application Lifetime, Visible, and Foreground resulting in calling requestLocationUpdates() twice (lines 16 and 33). Providing GPS Print’s services only require to call requestLocationUpdates() in onResume(). Starting the GPS already in onCreate() wastes energy, due to early binding. Releasing the GPS in onPause() (line 29) is timed correctly.

The reworked code is shown on the right of Figure 4. The energy code smell, marked red and boldface, in line 16-17 (left) has to be deleted in the refactored code in line 16 (right). Since initializing the GPS was already correctly realized in onResume() (lines 33-35) no code removal is required, here. Otherwise, the green and italics marked snippet has to be included.

Detecting and resolving energy code smells addressing binding resource too early is described in the following sections. Another flaw in GPS Print’s implementation is caused by not validating, if the GPS is active, before calling requestLocationUpdates() in onResume(). This issue is not included in the binding resource too early refactoring.

3.1 Detection

Detecting energy code smells is the first step for removing them. For analyzing source code, a simple GReQL evaluator is used. The query in Figure 5 calculates all classes calling requestLocationUpdates() in onCreate(), directly or indirectly. Those classes are candidates for restructuring, as switching on the GPS may be postponed to activating onResume().

The FROM clause maps the variables onCreate, caller, actClass, superClass, and callee to their according node types MethodeType, Class, and DataObject, defined in the SOAMIG Java metamodel (cf. Figure 2). The WITH clause defines conditions which must be fulfilled by the query result. This includes comparing attribute values like onCreate.name = "onCreate", testing for the name attribute of method onCreate and path expressions like onCreate <--{frontend.java.DataObjectHasType} <--{frontend.java.HasMethod} actClass ensuring the definition of method onCreate in class actClass. In this case, a method with name onCreate is searched. The class in which the method is located must inherit from class android.app.Activity. The concrete resource examined is requestLocationUpdates. According to the Java metamodel onCreate is approached from two sides. The REPORT clause defines the presentation of query results. Here, the names of a
Figure 5: Binding resources too early

calling class and the called method, which acquires the GPS sensor is reported back. The result is presented in the right upper panel of Figure 5. \((\text{GpsPrint, onCreate})\) indicates that \texttt{onCreate} in \texttt{GpsPrint} probably defines an early binding of a GPS sensor. A more comprehensive introduction to GReQL can be found at \cite{ERW08}.

### 3.2 Restructuring

Restructuring source code is the second step for removing energy code smells. For the presented binding resource too early-example, the restructuring was carried out manually to show its feasibility. Future activities will address removing energy code smells by utilizing graph transformation. It must, however, be noted that application behavior must not be changed after restructuring. Therefore, user interaction is needed to decide whether to accept a proposed restructuring or not, to keep the intended semantics. Different graph transformation languages like ATL (Atlas Transformation Language) \cite{JK06}, QVT (Query View Transformation) \cite{Kur08}, GReTL (Graph Repository Transformation Language) \cite{HE11} and others exists. GReTL is the likely choice for this use case, because it is part of the TGraph tool chain and has simple Java API besides GReQL used for graph querying \cite{HE11}.

Figure 6 shows an excerpt of the TGraph, representing the code of the GPS example depicted in Figure 4 before the restructuring. The complete graph conforms the Java metamodel (Figure 2) and contains 14880 nodes and 9034 edges.

The graph shows the \texttt{GpsPrint}-class (node v3488), which specializes the \texttt{Activity}-class (node v8676). \texttt{GpsPrint} defines two methods \texttt{onCreate()} (node v3644) and \texttt{onResume()} (node v6212). Following the CallsMethod-edges e5833 and e10467, both methods call \texttt{requestLocationUpdates()} (node v3644).

The binding resource too early-refactoring requires to delete \texttt{onCreate()}’s call of \texttt{requestLocationUpdates()}. So, restructuring results in deleting the CallsMethod-edges e5833 (marked red in figure 6) and adding another CallsMethod-edge connecting
onResume() to requestLocationUpdates(). The later was already correct in the original system (cf. green marked e10467), so no further activities were required here. An appropriate unparsers provides transformation services to translate these graphs to Java.

By using the Android simulator [Goo12] and by deploying the reworked GPS Print on an Android smartphone, it was shown that both programs behaved equally. So the refactoring improved the energy consumption but did not alter the program semantics.

4 Energy Code Smells

The previous section demonstrated by example how a specific energy code smell can be detected and removed by using refactoring. As a prerequisite, this approach requires the identification and cataloging of energy code smells. This catalog introduces a list of general energy code smells, which have to be adapted to be used on a specific platform. The proposed energy code smells are cross-platform as the underlying concepts and mechanisms stay the same on every platform.

Fowler [FBB+02] presents a catalog of code smells indicative of bad software design, described as a list of generic patterns. This paper envisions a similar compilation of energy-wasteful code patterns. Even though Fowler’s refactorings are not aimed at enhancing energy-efficiency, some “classic” code smells also indicate energy-inefficient code, one being dead code, for example.
Surveying literature, an initial set of energy code smells was identified. *Binding resources too early* was the subject of Section 3. The energy code smell *releasing resources too late* is basically the same: it describes program behavior wherein a resource is kept active, even though the application is not active itself. Further energy code smells are: *loop bug, dead code, in-line method, moving too much data, immortality bug, and redundant storage of data*. They are described in the following, each with a brief statement of the subject matter and motivation, explaining their energy inefficiency, and an account of the energy code smell’s detection and removal.

4.1 Loop Bug

*Loop bugs* [PHZ11] represent a program behavior wherein an application is repeating the same activity over and over again, without achieving the intended results.

A loop bug might occur due to external events, e.g. crashing of a server. As a result the application is trying to contact, the server, causing in repeatedly polling without receiving an answer, while using up energy for data connection. Programming mistakes can, also cause loop bugs, e.g. if an application is running in an (infinite) loop or descents into recursion, unnecessary enabling a device in every iteration, which is not actually needed. The loop might be running unnoticed by the user in background and drain the battery over time.

**Detection**
Detecting loop bugs requires to identify loops which always return to the same, initial system state, using energy-consuming components in the process. Exception handling in loops may be indicative of loop bugs, e.g. when a connection timeout exception is caught to retry and contact a remote server again.

**Restructuring**
One way to deal with loop bugs would be to introduce a maximum number of iterations, e.g. trying to poll a server three times before giving up (and possibly report the problem to the user).

4.2 Dead code

*Dead Code* [CGKO97] is source code which is never used, but needs to be loaded into memory and thereby consumes energy.

Some forms of dead code can be detected and removed as a compiler-level optimization. A variant of dead code is code which is invoked, but whose results are immediately discarded. These more “intricate” cases of dead code are easier analyzed and detected on source code level.

**Detection**
This energy code smell can, in the simplest case, be recognized by locating methods which are never actually called. Static code analysis is capable for detecting e.g. simple “left-overs” from previous maintenance activities like unused classes. More involved analysis is required to detect methods which are called under conditions which never arise at runtime,
or whose results are never actually used. Here, dynamic code analysis might help, keeping in mind that these analyses never guarantee a complete coverage.

**Restructuring**

If *dead code* is detected, it can simply be discarded completely.

### 4.3 In-line method

Da Silva and Brisolara [dSB10] have shown that *method in-lining* – replacing a method call with the actual body of the called method – may save energy, as the computational overhead of a method call is avoided.

*In-line method* is the opposite of *extract method* [FBB+02] defined as a technique to reduce duplicated code, and increase code maintainability. Applying this refactoring may therefore decrease the code’s maintainability and readability.

**Detection**

One approach to find method calls whose in-lining would have a high impact on energy-efficiency uses dynamic analysis, i.e. profiling the running application and record how often each method call occurs. This information can be used to select candidates for refactoring by setting a threshold of the number of invocations. Each method call in the code exceeding this threshold should be in-lined. In a similar vein, short methods, only containing a few lines of source code are reasonable candidates for further *In-line method*-refactorings.

**Restructuring**

Candidate method calls are replaced with the body of the called method, replacing the method’s parameters appropriately in the process.

### 4.4 Moving too much data

*Moving too much data* [HB10] represents unnecessary communication between processor and memory. If multiple parts of applications or different applications access the same set of data, it is usually called and stored several times, for example in database systems. Therefore, the data is written and pulled back from memory again and again.

Höpfner and Bunse show that moving data can be more expensive than recalculating it. In environments where much data is processed, the amount of energy consumed by storage units is higher than the amount consumed by CPUs [BH07]. In addition, if even two separate memories need the same data which was calculated before, it is not useful to store the data in lower level memory between the two calls. In most cases it is better to spend more instructions or to use higher level caches. For larger sets of data it is even possible to prefer cloud based storage solutions over local ones.

**Detection**

Methods loading data which have been saved previously by another method have to be detected, e.g. by querying for corresponding read and write methods.

**Restructuring**

To reduce data movement, refactorings similar to *In-line method* (Section 4.3) can be used:
in this case, methods reading data need to be replaced by the method calculating the data that was to be retrieved.

### 4.5 Immortality Bug

The *Immortality Bug* [PHZ11] describes an application’s respawning after explicitly being killed by the user.

Two different cases can occur: an application might spawn another instance of itself when receiving the kill-signal from the operating system, or another application is monitoring it, and restarts it immediately after it was killed.

**Detection**

The self-respawning of applications when being killed can be detected in a similar manner as *Loop Bugs* (Section 4.1): instead of catching, for example, a server response timeout, self-respawning applications handle the notification of being killed by “retrying” indefinitely. Detecting applications being restarted by other processes seems only feasible if source code of all applications involved, is available for co-analysis. Furthermore, dynamic analysis on system level might also indicate repeatedly starting and closing applications.

**Restructuring**

The code causing the spawning of another instance when being killed needs to be removed. Alternatively, refactoring could introduce persisting the number of previous restarts and have it compared to an upper bound of consecutive restarts, after which the “restart loop” is broken.

### 4.6 Redundant storage of data

*Redundant storage of data* [TDD+04] is a program behavior wherein different methods of an application store the same data in memory, instead of sharing it. Reducing this additional memory access decreases energy consumption.

**Detection**

To detect *redundant storage of data*, data-storing methods have to be identified, and compared to each other, to find those which store same data.

**Restructuring**

Unnecessary data accesses must be reduced and then the methods which store the same data can be combined. For this, the program behavior must be known.

### 4.7 Using expensive resources

*Using expensive resources* [SH11] represents possibilities to swap energy-expensive resources against “cheaper” alternatives, e.g. approximating global positioning data using WiFi (with access points whose position is known), instead of GPS. A prerequisite for this optimization is, of course, knowing the energy consumption of different resources for a given device.
Such a refactoring might impact the service quality of an application, like the higher accuracy of GPS versus approximating position data using WiFi, and should therefore only be applied if this higher service quality is not actually required. Similarly, resources not operable in certain situations, like UMTS or GPS in long railway tunnels might be switched off.

Detection
Usage of resource like GPS can be detected with appropriate queries. If an energy-expensive resource is found, it must be verified whether it would be possible to replace it with another, cheaper resource. For some applications, e.g. car navigation, WiFi position information might suffice (if available), while pedestrian navigation might require the finer-grained position information of GPS. Further detection might require additional information of the current surrounding of a mobile device.

Restructuring
The interface of the original resource can be implemented by a wrapper, using the cheaper, alternative resource or make use of information about the surrounding physical environment. Provided with such an implementation, the refactoring only needs to replace the original resource with the interface-compatible alternative.

5 Conclusion and further Research Opportunities

The paper motivated the application of software reengineering services for improving the energy consumption of applications. The binding resource too early-refactoring was defined and applied as one example for reengineering services, facilitating more efficient energy consumption. Applying the reengineering reference framework also showed, that existing tool support from software maintenance seems worthwhile for the development of energy-aware software. Further energy code smells and corresponding restructurings were sketched to motivate the development of a catalog of energy refactorings.

In summary, detecting energy code smells was shown to be possible via GreQL. In this paper, the actual restructuring was done manually. Besides the expansion of the catalog of energy refactorings, further research will address the extension and adaptation of the TGraph-based analysis and restructuring framework, so that the restructuring can be automated. Further validation of the presented techniques and an estimation of possible energy savings is still required to show advantages and limitations of the presented approach. In addition, the benefit of the specific refactorings has to be demonstrated. To make a quantitative statement a measurement platform with per application measurement capabilities is needed.

Preliminary results and related work in the area indicate that substantial energy-efficiency optimizations are possible on source code level, whose application will be eased by a structured, refactoring-based approach, and made readily available in a catalog of energy code smells. This approach can also be widen to applications on servers and desktop-pcs. The described energy code smells can be adapted to different areas, only the terminology will be changed but the process will be the same, like presented in this work.
References


