Fakultät Medien
Studiengang Mediensysteme
Professur Content Management und Web Technologie
Bauhaus-Universität Weimar

Diplomarbeit

Optimizing Abstract Data Types in Embedded Applications at Modeling Level

Eingereicht von:
Anne Keller

Eingereicht zur Erlangung des akademischen Grades
Diplom-Medienwissenschaftler.
30. November 2006

Erstgutachter: Prof. Dr. Benno Stein
Zweitgutachter: Prof. Dr. Bernd Fröhlich

Erklärung

Hiermit versichere ich, dass ich die vorliegende Diplomarbeit selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe. Die Arbeit wurde in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt.

Leuven, den 30. November 2006

Anne Keller

Zusammenfassung

Eingebettete Systeme sind spezialisierte Systeme, die vordefinierte Aufgaben mit speziellen Anforderungen erfüllen. Da das System hauptsächlich einer Aufgabe zugeschrieben ist, hat der Designingenieur die Möglichkeit Optimierungen basierend auf spezialisiertem Wissen durchzuführen, die maßgefertigte Lösungen liefern. Niedriger Energieverbrauch und hohe Rechenleistung sind valide Optimierungsziele, die die Mobilität eines Systems begünstigen.

Bis zu diesem Zeitpunkt werden Optimierungen eingebetteter Systeme traditionell am Quellcode durchgeführt. Da sich der Fokus auf dynamische und komplexe Applikationen verschiebt, ist dies nicht mehr ausreichend. Daher müssen diese traditionellen Ansätze mit Techniken, die Dynamik erkennen und diese in das Design einbeziehen, ergänzt werden.

Die vorliegende Arbeit stellt Optimierungen auf Softwaremodellebene vor, die eingebettete Softwaredesigns erzeugen, welche dynamische und Anwendungs-spezifische Informationen einbeziehen. Im Besonderen wird gezeigt, wie diese Transformationen sicher angewendet werden können und welche Kriterien dem Designer helfen, Transformationsmöglichkeiten zu erkennen. Um gleich bleibendes Verhalten der Anwendung zu sichern, werden Vor- und Nachbedingungen spezifiziert. Die Vorbedingungen garantieren, dass eine Transformation sicher angewendet werden kann. Die Nachbedingungen versichern, dass eine Transformation korrekt angewendet wurde. Sowohl Vorbedingungen als auch Nachbedingungen sind in der Object Constraint Language (OCL) ausgedrückt.

Schließlich werden mit der Umsetzung eines realistischen Beispiels Abstimmungen zwischen Datenzugriffen und Speichergebrauch, die Einfluss auf den Energieverbrauch des Systems haben, gezeigt. Die Ergebnisse sind viel versprechend und belegen die Anwendbarkeit von Softwareentwicklungstechniken im Bereich eingebetteter Systeme.

Abstract

An embedded system is a special-purpose system, which performs predefined tasks, usually with very specific requirements. Since the system is dedicated to a specific task, design engineers can optimize it based on very specialized knowledge, deriving an optimally customized system. Low energy consumption and high performance are both valid optimization targets, which increase the value and mobility of the final system.

Until now, software optimizations for embedded systems are traditionally performed on the source code. However, as the focus shifts to dealing with highly dynamic and complex applications, this is not sufficient anymore. Therefore, traditional optimization approaches need to be supplemented with techniques that detect dynamism and incorporate it into the design.

The scope of this work is to introduce optimizations at the software modeling level that are capable of producing embedded software designs, that incorporate dynamic, application specific knowledge. Specifically, it is shown how to express these transformations so that they can be applied safely (i.e., behavior preserving) and what criteria can help the designer to detect transformation opportunities. To ensure preserved application behavior, pre- and postconditions are specified. The preconditions guarantee that a transformation can be applied legally, while the postconditions ensure that the transformation was executed correctly. Both preconditions and postconditions are expressed in the Object Constraint Language (OCL).

Finally, with the implementation of real life case studies it is possible to show that trade-off points, in terms of memory footprint and data accesses can be identified, which have an impact on energy consumption. These results are promising and validate the feasibility of implementing software engineering techniques in embedded systems.

Acknowledgments

I especially want to thank Marijn Temmerman for her valuable advice and guidance during the work on this thesis, and for always taking time to have one of many discussions.

Also many thanks to Prof. Francky Catthoor and Sven Meyer zu Eissen for their much appreciated feedback.

I would like to thank Alexis Bartzas, Miguel Peon, Patrick Riehmann and Stelios Mamagkakis for reading and commenting on this thesis and all the help they provided.

And finally, big thanks to everyone, who made my time at IMEC a bliss.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Introduction

Unlike general purpose computers, embedded systems are highly constrained by their resources (i.e., battery life, processing power, memory usage). High energy consumption drains the battery and is thus a key factor for the mobility of a system.

When mapping software to an embedded system, it is crucial to respect those constraints. In particular, modern data intensive applications with large, heavily accessed, dynamic data structures pose a problem to the energy consumption of a system. Accesses to large memories translate directly to high energy consumption. (This is specifically true for SRAM on-chip memories as used for caches and scratchpad memories.) Therefore, memory accesses and memory footprint are valid and important cost metrics, which need to be decreased in order to optimize embedded software designs.

In the embedded system community both methodologies and tools are developed to map software efficiently to embedded systems, decreasing memory accesses and memory footprint. In the data structure optimization domain this is, among others, done by intermediate variable removal [16], reducing size per element, reducing the number of elements, and reducing traversal of elements [4]. All this happens at source code level and yields local optimal solutions. To find more global solutions, however, the complexity of mapping and analyzing larger, highly dynamic and often accessed data structures is very high.

Thus, raising the abstraction level (i.e., from concrete data structures and data types to abstract data types – ADTs) in order to gain a broader view of the application's data usage is necessary and has clear benefits.

ADTs (cf. Appendix A) are a method of abstraction of the complexity that is associated with data organization. For example, instead of reasoning about whether a Stack is implemented with arrays or lists and how those are implemented on a certain architecture, the Stack ADT provides a Stack as data entities, on which the operations *push* and *pop* can be performed.

By hiding underlying complexity, it is possible to reason about dynamic, application specific knowledge and obtain less localized solutions. This means that complexity, such as memory allocation, etc., is hidden, yet embedded platform relevant cost

metrics are used.

In this work, reasoning about ADTs is done at the modeling level. This means, models of ADTs are analyzed and transformations to optimize those models are suggested. These transformations yield solutions that can be considered as trade-off points between the chosen cost metrics, memory footprint and data accesses. Thus, an embedded system designer uses this catalogue of transformations to implement at design-time an optimal trade-off point for a certain metric.

A clear contribution of this work is the extension of the catalogue of transformations presented in [31], [32] and [33]. These transformations are created using 2D- and 3D-Graphics applications. In this thesis, the feasibility of specific transformations for a different problem domain, namely text processing, is shown.

Further, this work tries to formally specify model transformations according to *transformation definitions*, which are composed of *transformation rules* (cf. Appendix A and [15]), thus gaining more insight into the transformations, as well as demonstrating the feasibility of future automation. Automatic model transformations enable the integration of the transformation catalogue into a refactoring-like tool. Here the designer can, based on his domain knowledge, choose from a list of transformations, which will be performed on the model. Thus, the error prone and tedious work of transforming a model by hand and keeping different diagrams consistent ([17]) is taken away. Such a tool creates optimized models, from which code is generated that can then be subject to further optimizations.

Finally, this work presents concrete profiling results in terms of data accesses, memory footprint and energy consumption, that show the feasibility of the optimization transformations in the embedded systems domain.

Related Work

This work is set between two domains. On the one hand, embedded systems constraints are the driving force behind the proposed transformations. On the other hand, the transformations are defined with software engineering techniques. In the following relevant related work from both domains is presented.

In the embedded system domain UML (Unified Modeling Language) is a known and widely used modeling language. Especially in real-time systems UML is used to model timing critical aspects. [9] presents a UML profile for embedded real-time systems. However, the focus is on time, performance, and quality of service, rather than on memory usage and energy consumption.

In further usage of UML in embedded systems, the general focus is on modeling the hardware platform with structural and behavioral models, as introduced in [19] for SoC (System-on-Chip) platforms. For example, typical concepts in this context are ‘channel’, ‘port’, ‘clock’. The work in this thesis is complementary, because it considers embedded systems design at a higher level of abstraction.

Also model-based development is mentioned in the embedded system community. [28] argues that raising the level of abstraction, i.e., to model-based development, is applicable for embedded systems. Complex development steps can be described through both process and product models. But here again, the suggested product models focus more on the embedded platform and not on high-level concepts.

Until now, optimization on data and memory usage of embedded systems is not done at the platform-independent UML level. However, there are a many approaches that focus their optimizations on the code level, as in [6], [8] and [35].

On the other hand, this work is influenced by the software engineering domain. Here, especially the Refactoring and Patterns communities are relevant.

[20] introduces a wide range of patterns for small memory systems. The suggested patterns range from patterns for memory allocation and small data structures to secondary storage and compression patterns. General concepts like compression are used in this work as well but the focus is rather on the guided exploration of solutions with ADT models.

Literature on refactoring is very close to the transformations applied in this work. Existing code ([22], [26], [10]) or UML models ([29], [13]) are transformed stepwise, while the behavior is preserved. However, the main motivation of these works is to optimize the object-oriented design, in order to allow software evolution.

Also the model-driven engineering domain is relevant. Especially lately a lot of approaches and concepts are explored to transform between models in different languages, as well as between models in the same language. Both [7] and [18] give classifications for various model-driven approaches and concepts.

Overview

The remainder of this thesis is structured as follows. In Chapter 2, all transformations are introduced. A general motivation including a short description of each transformation is followed by a specific example application of the transformation on a driver application. The description of each transformation concludes with high-level estimates for the considered low-level cost metrics.

In Chapter 3, the theoretical background of model transformations is laid out. Here it is shown, how a transformation is split into atomic transformation steps and how these steps are formally described as transformation rules. Further, it is shown how behavior preservation is demonstrated at modeling level. This again is presented together with an example.

In Chapter 4, the experimental results of implementing various transformations are shown. In this chapter, concrete numbers for the cost metrics, memory footprint, data accesses, and energy consumption are shown. Further, trade-off points between those cost metrics are identified.

This thesis closes with Chapter 5 concluding and pointing towards future work in the field of ADT transformations for embedded systems.

Chapter 2

Transformations Introduced

In this chapter the model transformations that are subject of this work are introduced. The chapter starts with an introduction of the driver application, in order to then introduce and motivate the suggested transformations. Examples show how the aforementioned application specific knowledge largely steers the transformations. Each section, which corresponds to one transformation, is finished with a theoretical analysis of the trade-off between memory footprint and data accesses. While in Chapter 4 the assumptions are verified through profiling the original and the transformed application code, here the performance estimates use high-level estimates based on a reference architecture.[1]

2.1 The Design Space

One objective of this work is to further explore the design space of model transformations in the embedded systems context. This should give greater insight into the problem domain, specifically, reveal new and different solutions, which are obtainable in a guided fashion.

Figure 2.1 shows possible And-Or-Graphs depicting decisions that lead to specific transformations. This graph is used to showcase new transformations and to reason about the possible transformations.

The creation of the design space is twofold. On the one hand, model elements are determined that can be subject to transformations. These are *Container*, *Part*, *Relationship*, *Operation* and *Attribute*. On the other hand, basic techniques that influence the trade-off between the cost metrics, memory footprint, and data accesses, are identified and assigned to the different model elements. This process is complicated by the fact that the model elements are not independent of each other. Some are contained in each other and the contained ones can not exist without its container (e.g., *Container* holds *Attributes*, and *Operations*). Others are substitutable (e.g., an *Attribute* can be expressed as a *Part* with an association to the former containing *Container*).

[1] These high-level estimates are based on the assumption that each data access corresponds to one memory access. Further, the following sizes are assumed for data types: 1Byte - Char, 4Byte - Pointer, 4Byte - Integer.

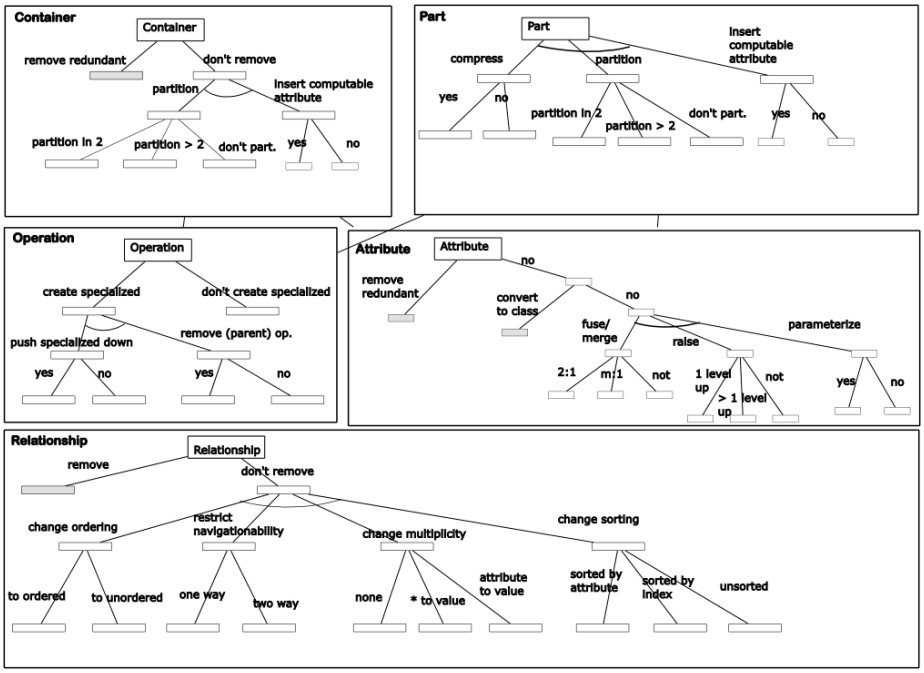

Figure 2.1: And-Or-Graph for the Design Space of ADT Model Transformations

The basic techniques are insertion or deletion of a model element. Insertion and deletion can be applied to all elements, while, here again, the dependencies between different model elements are important. Further, techniques for data handling are compression and partition. Any compression decreases the memory footprint, but makes accesses more costly (due to decompression). Partition on the other hand 'spreads' data entities and groups them in different partitions or contexts. This is beneficial for data accesses, since – intuitively – data is easily accessible (i.e., due to smaller clusters) and grouping resembles a kind of sorting, which makes search and traverse operations less costly. These techniques are applied to all model elements holding data entities (i.e., *Container*, *Part*, and *Attributes*). Further, all model elements have element specific transformations that are explained only through the definition of each model element. For example, changing the navigability of a relationship is *Relationship* specific.

The design space depicted in Figure 2.1 lacks a concrete ordering of decisions and dependencies. This is part of the work of Marijn Temmerman, who currently pursues a PhD at IMEC vzw., [32]. An example of the methodology, that is applied in her work, to define the design space and derive solutions from it, can be found in [3]. In this thesis however, sample transformations are taken from the design space and are applied on real life examples. The transformations are theoretically investigated, implemented and profiled to show the correctness of the assumptions made.

Figure 2.2 shows the concrete transformations ordered according to the model ele-

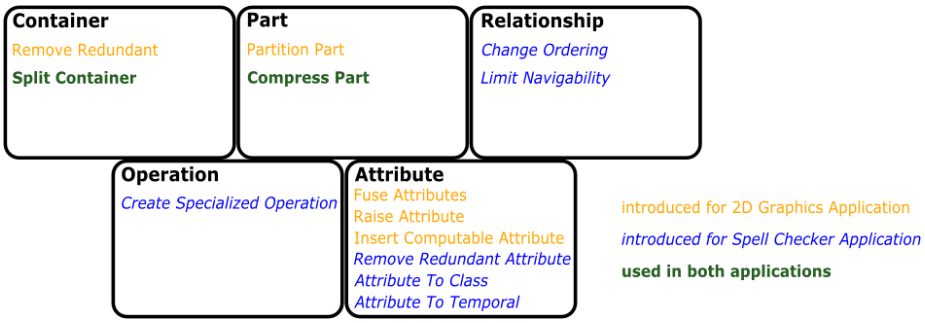

Figure 2.2: Transformations ordered by Model Elements

ment that they can be applied on. Transformations in *italic* font are newly created for the spell checker driver. The transformations in lighter colored font are introduced in [32] and developed for the 2D-Graphics domain. The transformations in **bold** are also introduced in [32] and developed for 2D-Graphics applications. However, they are also used on the spell checker application.

2.2 Driver Application – The Spell Checker

The driver application used in this work is a spell checker.

The spell checker receives words as input and verifies whether they are spelled correctly. In order to do so, it compares the input word with a internal list of words, the dictionary. The output is a list of all unknown, therefore misspelled, words.

The internal dictionary is created upon startup of the application. Therefor, a list of words is read and stored in a Trie ADT. The Trie is a space efficient abstract data type (cf. Appendix A), which facilitates fast look up of a word.

Trie

Figure 2.3 shows a Trie. In the example, the dictionary consists of only four words, ‘boot’, ‘tea’, ‘the’ and ‘zoo’. Each node of the Trie has a value, which is a letter. Additionally, two pointers and an *end* flag (the *end* flag is not shown in the picture) are stored in each node. The pointers act as edges of a tree, which connect letters to a word and words to the complete dictionary. The *down* pointer points to the next level of possible letters (i.e., connects letters to a word) and the *next* pointer points to the following letter on this level (i.e., connects the complete dictionary).

To search a word, the Trie is traversed starting at the first level, searching for the desired letter. If it is found, the *down* pointer is followed to the next level. There the desired second letter is searched. This repeats until a letter is not found (i.e., the word is not in the dictionary) or a set *end* flag is encountered together with the end of the input word (i.e., the word is found).

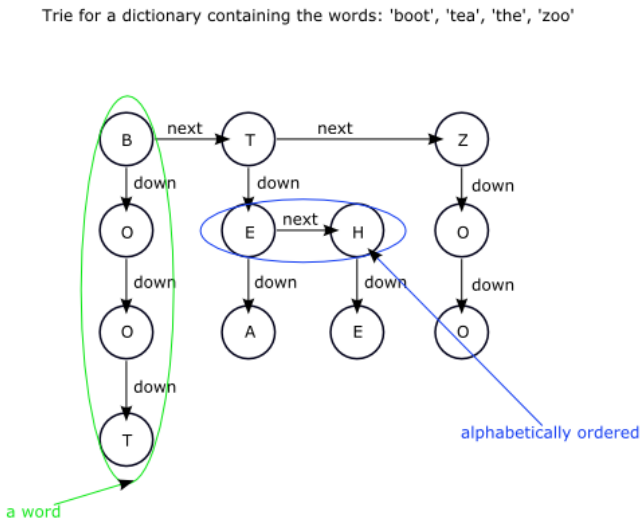

Figure 2.3: Dynamic Data Type Trie

2.3 The Initial Application Model

All proposed transformations in this work are performed on models. Models can be early entities of a design flow or representations of already existing applications. If, in the latter case, the creation of the model from the application is done manually and not with the help of a Re-engineering tool, it is important to capture both functionality and design intentions in the model.

Figure 2.4 shows the initial UML class diagram of the spell checker model that is used in the remainder of this work. This model can be understood as an entity in the beginning of a long design flow towards an optimized embedded software application.

The model consists of five classes. The central class, which steers all functionality, is *DictionaryEngine*. It triggers the creation of a new dictionary by calling *initDict()* on *Dictionary*. To check a word, the word is read from *Textstream* and sent to *Dictionary*, where the check is performed.

Attributes of *DictionaryEngine* are *WordBuffer* and *MisspelledList*. *WordBuffer* holds the currently read word from the text stream. *MisspelledList* is a list of all misspelled words that is extended, whenever a word is not found in the dictionary.

As briefly mentioned above, *Textstream* represents the input of words into the dictionary, both for the initial creation of the dictionary and for the lookup of a word in the dictionary. Thus, *Textstream* can represent an input file containing words, a wordlist or any other kind of input imaginable.

The dictionary is represented by the remaining three classes *Dictionary*, *Word* and *Character*. For the following reasoning about ADT transformations at modeling level, it is especially important to understand, how the Trie ADT is present in the model in Figure 2.4.

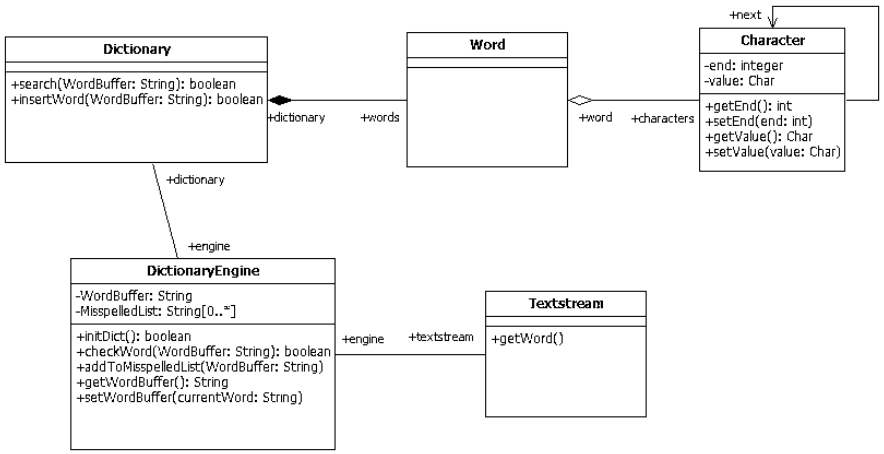

Figure 2.4: UML Class Diagram of the Initial Model

Dictionary can be seen as representing the whole Trie ADT (as shown in Section 2.2). Here the operations for creating the dictionary and searching a word reside. *Dictionary* is a container for words. The strong aggregation relation between *Dictionary* and *Word*, states that *Dictionary* consists of words. The words are ordered within *Dictionary* (in our case we assume alphabetical ordering) and each word exists only once.

Word is a container for characters. Here the weak aggregation between *Word* and *Character* is used to indicate that the characters can be reused for different words, as it is done in the Trie. This relationship also represents the *down* pointer found in the Trie. *Word* does not have any attributes or operations. It is important nonetheless, since it emphasizes the functionality and structure of the dictionary.

Strictly speaking *Character* represents a node in the Trie. The *next* pointer, which exists in the Trie, is represented in the model as the *Character-Character* relationship. *Value* is the letter represented by the node and *end* is a flag indicating the end of the word.

In the following sections, the transformations are introduced ordered by the model element they are applied to.

2.4 Container

2.4.1 Split Container

Short Description and Motivation In this transformation a Container consisting of Parts is split into two Containers. The Parts are distributed among the two containers, this is shown in Figure 2.5. The Parts of the two Containers are a disjoint set.

The distribution of the Parts is steered by a splitting criterion, which takes the access

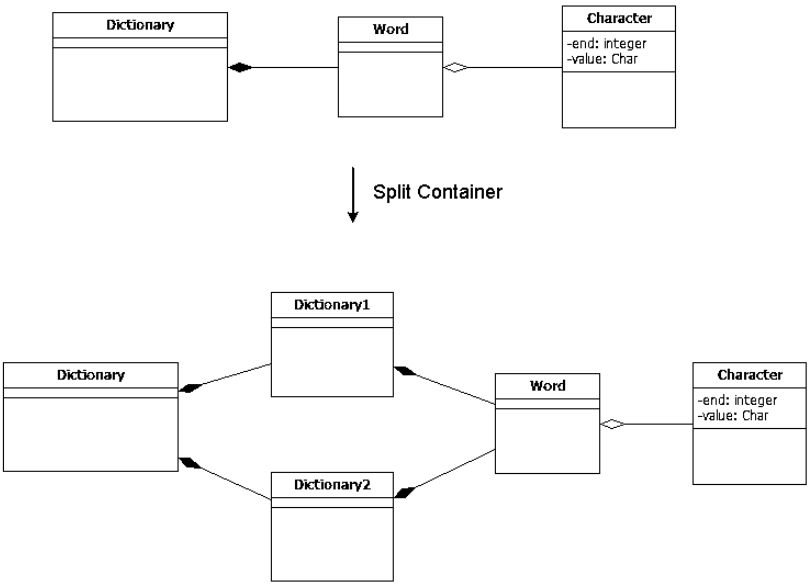

Figure 2.5: Split Container Transformation

behavior to the Parts into account. If Parts are accessed with different frequency, the splitting criterion assigns often accessed Parts to one Container and less often accessed Parts to the other Container. What values are exactly associated with 'often accessed' and 'less often accessed' depends on the designer and the concrete application.

To conclude, if Parts of a Container display very different access behavior, the Split Container transformation should be applied.

Dictionary Example In a spell checker such different access behavior, as described above, can be found in the lookup frequency of certain words. Since some words are more and some less commonly used in a language (E.g., in an English text the word 'the' is more often used than 'xylophone'), they are more and less often checked in the dictionary.

To make use of this information, the often accessed words are grouped into one Container and the less often accessed words into the second Container. This results in one relatively small, highly accessed Container of words and a larger, less frequently accessed Container.

High-level Estimates This transformation draws its gains from new implementations that are made possible and at the same time suggested through the new design.

The memory footprint can be decreased, if the large and less often accessed Container is further compressed. Through the compression more accesses are needed to

access a Part. But the trade-off between increased accesses and decreased memory footprint is expected to be in favor of the memory footprint, since the large Container is not accessed often. The overhead for decompression is less likely to be paid, because of the overall low access frequency to this container.

Further, this transformation can decrease the energy consumed of the application. This is based on the assumption that the two newly created Containers are put onto two different memories, each according to the Container's size. This yields one small, heavily accessed memory to which the accesses are less costly in terms of energy and a larger, less energy-efficient memory.

The concrete results for this transformation are presented in 4.2.3.

2.5 Part

2.5.1 Compress Part

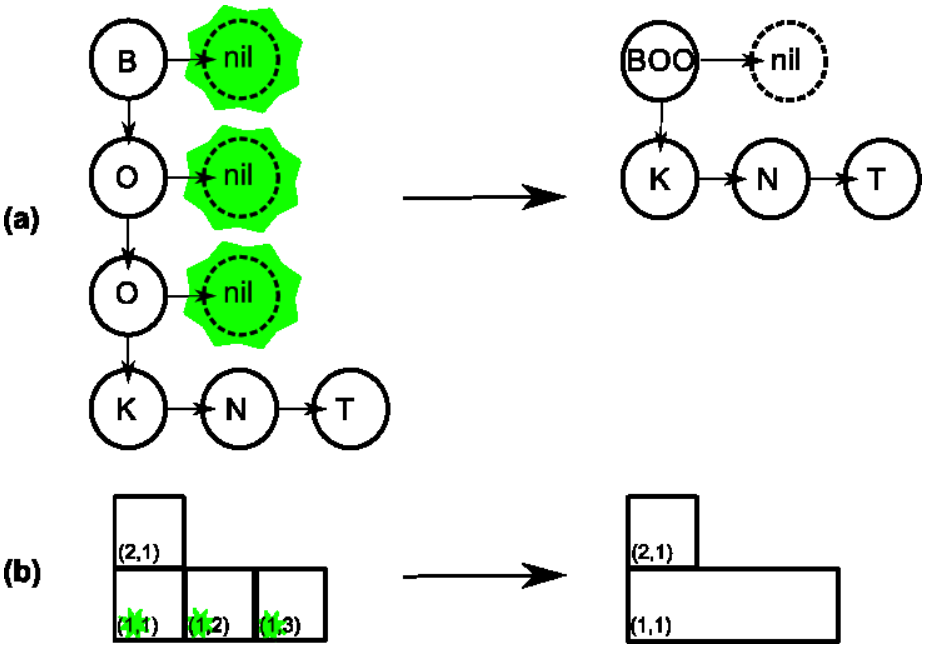

Figure 2.6: Compress Part Transformation – Example

Short Description and Motivation The Compress Part transformation groups several instances of one Part into one instance, thus creating a new Part. *This is applicable, if several Parts in one Container have the same value for an attribute.* Figure 2.6 shows

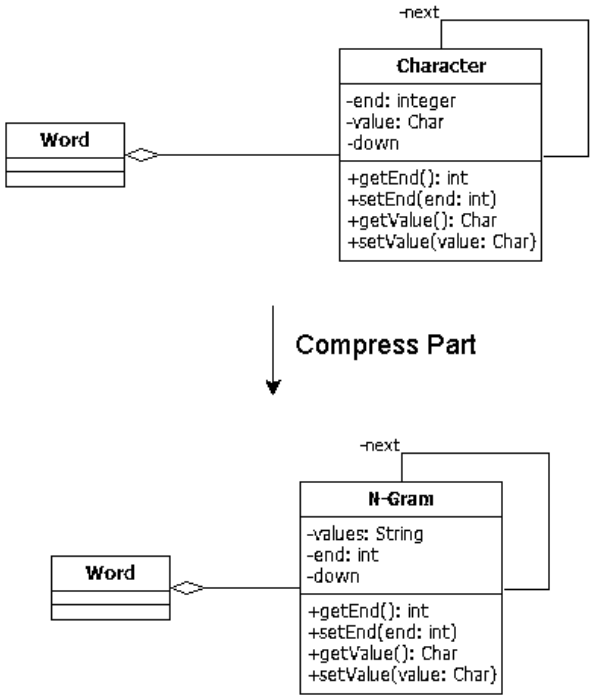

Figure 2.7: Compress Part Transformation

two examples motivating this transformation, one for a Trie and one for a game piece in a 2D board game.

Attention needs to be paid, if the Parts have several attributes with different values. In this case it needs to be assured that no information is lost through the compression. This can, for example, be done by storing a list of those different values in the newly created Part.

Dictionary Example Figure 2.6 illustrates under which circumstances parts can be compressed.

In the Trie example in Figure 2.6a three unary nodes containing 'B', 'O' and 'O' can be compressed to one node. This is done considering that all three have `nil` *next* pointers and thus, searching a word always takes the path through the sequence 'boo' (i.e., a suffix trie is created).

The example 2.6b shows a piece in a 2D board game. Here again, several pieces have the same value for one attribute. Three pieces lie on one level (i.e., have the same y-coordinates) and can thus be grouped together. Instead of saving each location individually, the starting and end points of the grouped piece are stored and a new Part

Segment is created.

Figure 2.7 shows a UML class diagram for the spell checker example. A new Part called *N-Gram* is created. Here the values of the single nodes are stored plus *end* and *next* pointer. *Value* is a list of all letters, *end* is set to true, if the last node is an end node, and *next* has the same value for all nodes (cf. Figure 2.6).

High-level Estimates This transformation is expected to decrease the overall memory footprint, since several instances of one attribute are compressed to one instance. A simple calculation shows that on the dictionary example.

Per node:
`1 Byte` for the *value* (char)
`4 Byte` for the *end* flag
`4 Byte` for each pointer (*next* and *down*)

`= 13 Byte` per node
`= 39 Byte` for three nodes

Instead of three nodes, only one node is stored:
`13 Byte` for one node
`2 Byte` for each of the two character values
`= 17 Byte` for one compressed node.

Overall data accesses are reduced, since not each value is accessed individually for traversal[2]. In the dictionary example, for looking up the word 'boot' three accesses are made to the node 'boo' (i.e., reading three character values), one following the *down* pointer plus accesses for finding the letter 't'. For comparison, before, two accesses were made to each node: one checking the value and one following the *down* pointer. Thus, the number of accesses is also reduced.

2.6 Relationship

2.6.1 Change Ordering

Short Description and Motivation In this transformation the ordering of the Part-Container relationship is changed. *It should be applied, if Parts in a Container are not accessed following their ordering.* Thus, the new ordering should express the access behavior of its Parts more precisely.

Figure 2.8 shows a class diagram after the transformation is performed. In UML 2.0 a constraint exists to indicate the ordering of relationships and attributes, the keyword `ordered`. This, however, leaves the interpretation on what ordering this includes to the reader of the diagram. Common sense is, to assume an ordering depending on the element's type (e.g., letters are ordered alphabetically). To dissolve these ambiguities,

[2]Note again, in the following one data access is counted as one memory access.

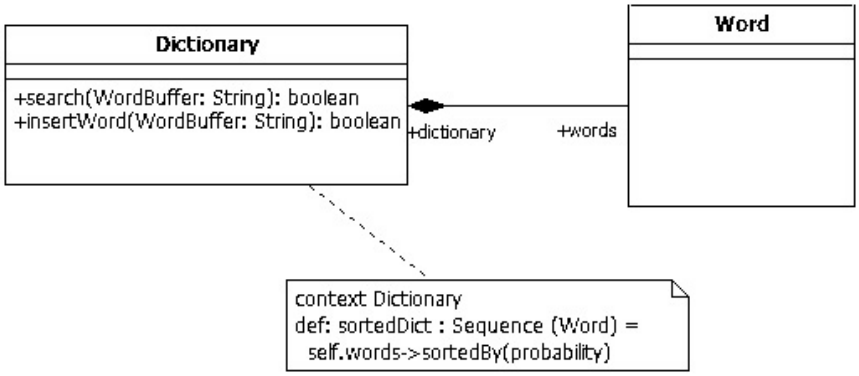

Figure 2.8: Change Ordering Transformation

the ordering is further specified through OCL (Object Constraint Language) constraints as seen in Figure 2.8. (A short introduction to OCL is given in Section 3.4.)

Dictionary Example In the dictionary example the Change Ordering transformation can be applied both to the words in a dictionary and to the characters in a word. In the following, changing the word order in the dictionary is applied and explained.

Words are alphabetically listed in the dictionary. In any alphabet-based language a frequency table for each letter can be produced. This means that some letters are more common in the language than others. According to this reasoning, words starting with the most common letter are more commonly used than words starting with less common letters. (This theory is without doubt arguable, but the presented results in section 4.2.5 support the thesis.) Therefore, the ordering of the words in the dictionary is changed from alphabetical to an ordering reflecting the letter frequency in the English language. (Again, more information and results can be found in section 4.2.5.)

Figure 2.8 shows the class diagram of this transformation. Here, the change in ordering is specified through an added OCL constraint. This constraint creates an ordered sequence of words (*sortedDict*) according to the first letter's probability in English.

High-level Estimates Changing the ordering of a relationship is not expected to have any impact on the overall memory footprint, since no data values are added or deleted.

However, the overall data accesses are decreased. A possible increase in accesses could result from the added complexity for insertion, due to a different ordering. This is not the case however, since in a list-structured implementation only the comparison operator (i.e., '<') is overloaded, in order to insert the words in the dictionary. The main decrease in accesses is expected for retrieval of the words, since here the average lookup is expected to be less costly.

Results for this transformation can be found in section 4.2.5 of Chapter 4.

2.6.2 Limit Navigability

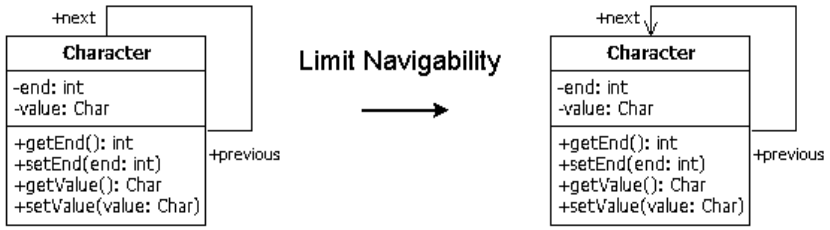

Figure 2.9: Limit Navigability Transformation

Short Description and Motivation The Limit Navigability transformation reduces the navigability between two classes from two-way navigable to one-way. *It should be applied, when two classes have a two-way navigable relationship but do not make use of it.* This means that only one class is calling the other.

The reason why this transformation is beneficial, is that for each direction in which a relationship is navigational, additional data about the target class needs to be saved. Thus, introducing overhead for storing, accessing and checking this data.

Dictionary Example In the spell checker example, this transformation is applied to the Character-Character relationship *next*. If it was two-way navigational, each character needed to store information about its predecessor and its successor. This is avoided by limiting the navigability explicitly to one-way. Now, through each character the following character can be accessed (i.e., the Trie is traversed downwards) but not vice versa, i.e., a word can not be read backwards.

High-level Estimates The overall memory footprint is decreased by limiting the navigability. On the spell checker example, this means that only one pointer, instead of two, needs to be stored.

The number of data accesses is also decreased. For example, this can be shown for setting up the dictionary. If the characters stored both data about their successor and data about their predecessor, then both values needed to be written at startup. Since this is not the case for limited navigability, the accesses (i.e., write accesses) are reduced.

2.7 Operation

2.7.1 Create Specialized Operation

Short Description and Motivation In this transformation specialized operations are created, to access Parts that are stored in different ADTs.

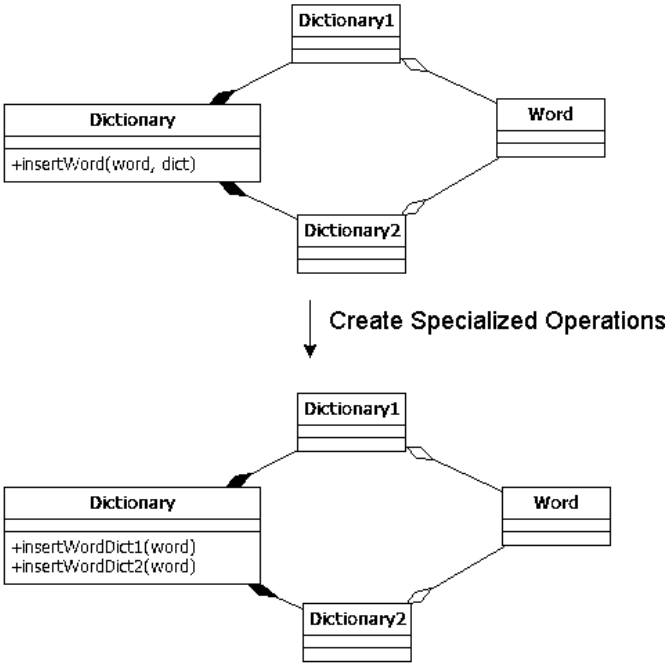

Figure 2.10: Create Specialized Operation Transformation

This transformation should be applied, when the model contains a hierarchy of Containers (as, for example, created in Section 4.2.3) and these containers store their Parts in very different ADTs. To avoid the overhead of accessing both Containers with a generalized method, new methods, specialized for each Container, are created.

Example In [10], this transformation is described in the 'Push Down Method' Refactoring. Although this refactoring is intended for classes in inheritance hierarchies, it is also beneficial for the case of an aggregation or composition relationship as discussed here.

Assuming that Dictionary1 in Figure 2.10 keeps its data in a Set ADT, while Dictionary2 keeps its data in a Bag ADT, a method that inserts Parts into those Containers can vary strongly in functionality. A Set ADT is a collection of data items, in which every item exists only once. A Bag ADT however consists of data items that can exist more than once in the Bag. Adding an item to a Set requires a check of all items to ensure that the item-to-add does not exist in the Set yet. Adding an item in a Bag

is easier however, since no checks need to be performed and the item can simply be added. A method that adds items into both Sets and Bags will at least include some overhead checking to determine whether the item should be added in the Set or the Bag ADT. Specialized operations however can be called directly (since it is known in which Container to add) and thus avoid this overhead. Figure 2.10 shows an example of this transformation.

High-level Estimates The overall number of data accesses is decreased by this transformation. Instead of calling *addElement(element, container)* and following a conditional in *addElement* determining what ADT a Container is, the specialized operation *addElementCont1(element)* is directly called. Therefore, there are gains for each insertion operation.

The overall memory footprint of the application does not increase through this transformation, since no data entities are created or deleted.

2.8 Attribute

2.8.1 Make Attribute Temporal

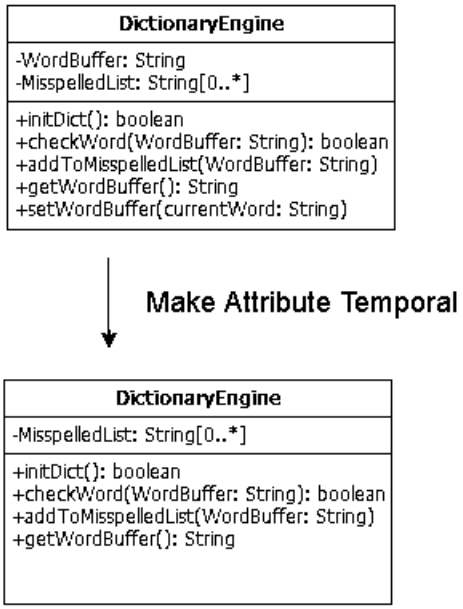

Figure 2.11: Make Attribute Temporal Transformation

Motivation and Short Description The Make Attribute Temporal transformation removes an explicit (possibly global) variable and instead adds it as input to methods that consume it and as output to methods that change it. If the program is executed, this attribute is created as a local temporary variable.

This transformation should be applied, if an attribute serves as a buffer or temporary variable rather than a long lasting storage and it is consumed right after its creation. A cue to this can be many reads and writes to one attribute through different methods.

Dictionary Example In terms of the spell checker example, such an attribute is found in *WordBuffer*. *WordBuffer* holds the current word read from the text stream and is used as input for the *insertWord* and *search* method. As soon as the value of *WordBuffer* is read from *Textstream*, it is passed on to one of the methods. Afterwards, a new word is read from the text stream and *WordBuffer* is written again.

Removing the attribute explicitly from the model is without side effects because the attribute is always consumed right after it is written.

High-level Estimates The overall number of data accesses does not change for this transformation, since the functionality of the application remains the same (i.e., the attribute is as often written and read as before).

However, the overall memory footprint is reduced considering the lifetime of the application. This is explained through the attribute being created on the stack now rather than the heap. When it is not used, i.e., not written or read, is does not take up space in memory.

2.8.2 Remove Redundant Attribute

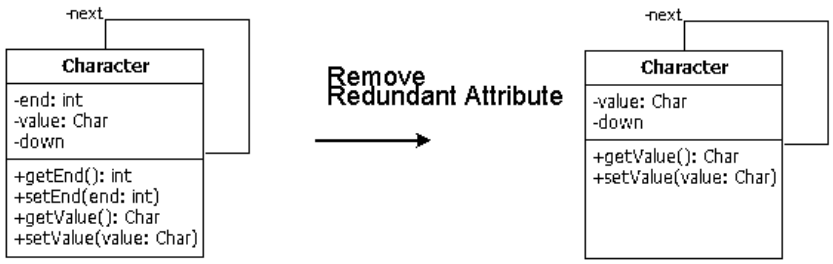

Figure 2.12: Remove Redundant Attribute Transformation

Short Description and Motivation As the name suggest, in this transformation redundant attributes are removed. *It should be applied, if the model contains an attribute*

that is redundant, because it represents unused, already current or implicitly retrievable information. If an attribute is indeed redundant, is only deducible with close insight into the application and its behavior.

Dictionary Example In the case of the spell checker *end* is the redundant attribute to be removed, as shown in Figure 2.12.

The information whether a character is a possible word ending is necessary, since only with this information a word can be correctly identified as misspelled. E.g., assume the word part 'appl', without checking *end* this word part would return true. But since 'l' is no possible word ending for 'app', it evaluates to false. And only 'apple' returns the correct results.

The *end* attribute is removed nonetheless. To denote the end of a word the *down* pointer is used to point to the node it originates from. This makes use of this pointer, since it is not used at the end of a word.

This algorithm has still one disadvantage however. Consider the words 'nut' and 'nutmeg'. If the *down* pointer was set to point to its originating node at 't', the word 'nutmeg' would never be recognized as correctly spelled. Therefore, in case of a word ending and a longer word with that suffix, the decimal ASCII value of the letter is divided by two. This yields symbols that are not part of the alphabet. If such a symbol is encountered, its decimal value is multiplied by two and the letter is used as normal. (An example: The decimal values of upper case letters range from 65 to 90, the decimal values of lower case letters from 97 to 122. 122 divided by two is 61, which results in '='.)

High-level Estimates The above explained algorithm introduces a new trade-off between storage and computation.

In the spell checker example the number of data accesses is reduced, this is due to several reasons.

For setting up the dictionary, instead of writing *end* for each node, only for the ending nodes the pointer is set or the new letter value is calculated.

For looking up a word accesses are only slightly increased. This is due to the fact that the number of suffixes that are also valid words (e.g., 'over') is quite small in the English language. Further, combined words – as in other languages – are not common in English. Therefore, the computational overhead for adding and decrypting such a node is limited.

The memory footprint is reduced, due to the deletion of an attribute. The attributes that substitute for the removed attribute, would be allocated in either case and thus do not increase the memory footprint.

Results for this transformation can be found in section 4.2.4.

2.8.3 Attribute to Class

Short Description and Motivation In this transformation an attribute is transformed into a separate class connected to the original class with an aggregation. Using an aggregation relationship is based on the characterization of aggregation with primary and

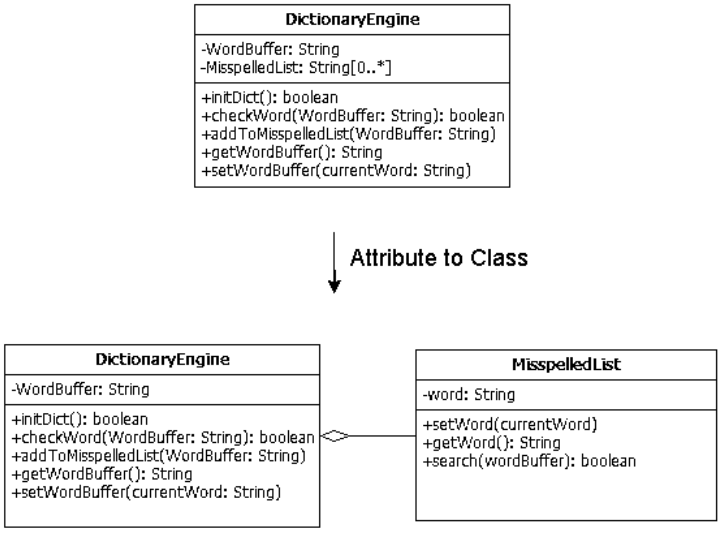

Figure 2.13: Attribute To Class Transformation

secondary characteristics introduced in [27]. *This transformation should be applied when a possibly large, complex and often accessed attribute is encountered.* In this case, transforming the attribute into a new class is believed to be beneficial, because a class can again be subject to various transformations.

Dictionary Example In the spell checker example, storing the list of misspelled words as a class (depicted in Figure 2.13), gives the chance to again transform this class. For example, the new list class can further be separated into words or characters to suggest different data structure implementations (cf. Split Container Transformation).

High-level Estimates The mere transformation of an attribute to a class is not expected to change the number of data accesses nor decrease the memory footprint. This transformation is however a starting point for many transformations that bring different gains and drawbacks, as presented above.

Results for this transformation can be found in Section 4.2.6.

Chapter 3

Transformations – Theoretical Background

In the this chapter, the formal basis of the aforementioned model transformations is laid, based on concepts and definitions from the Model Driven Architecture (MDA). Model Driven Architecture is a framework for software development defined by the Object Management Group (OMG). (See [15] for more information.) In the following the MDA definitions and concepts regarding model transformations and metamodeling defined in [15] are used. Based on this, the transformations in this work are identified as transformations between models of the same language. (I.e., both source and target model are expressed in the same modeling language.)

To describe the transformations thoroughly and to capture all important aspects, models at different abstraction levels are used. Figure 3.1 shows the different levels of abstraction. The Meta Level is used to define both the ADT metamodel and the transformations. At the Application Level, UML class diagrams show the static properties of the application, while UML sequence diagrams show the behavior in snapshots. At the Object Level, UML class diagrams and UML sequence diagrams are used to show the concrete objects, created in specific snapshots of the application. In the following, the term snapshot is used to describe the instances of data at a specific instance of time.

It is not in the scope of this work to examine transformations between these models, however. For now, the models that exist at different levels of abstraction exist in parallel, are instances of each other and are manually synchronized. Instead the different abstraction levels are used to capture knowledge, where it is relevant.

In the following these different abstraction levels are explained together with the kind of information they hold. The levels correspond to the four modeling layers used within the MDA framework. The highest layer, namely M3, is only shortly mentioned in this work and not explained in its own section. The naming of the layers is changed to a more suitable description within the context of this work. ‘Layer M3’ is renamed to ‘Meta-Meta Level’, ‘Layer M2’ to ‘Meta Level’, ‘Layer M1’ to ‘Application Level’ and ‘Layer M0’ is renamed to ‘Object Level’.

The chapter is finished with a full transformation definition for the Split Container

Transformation.

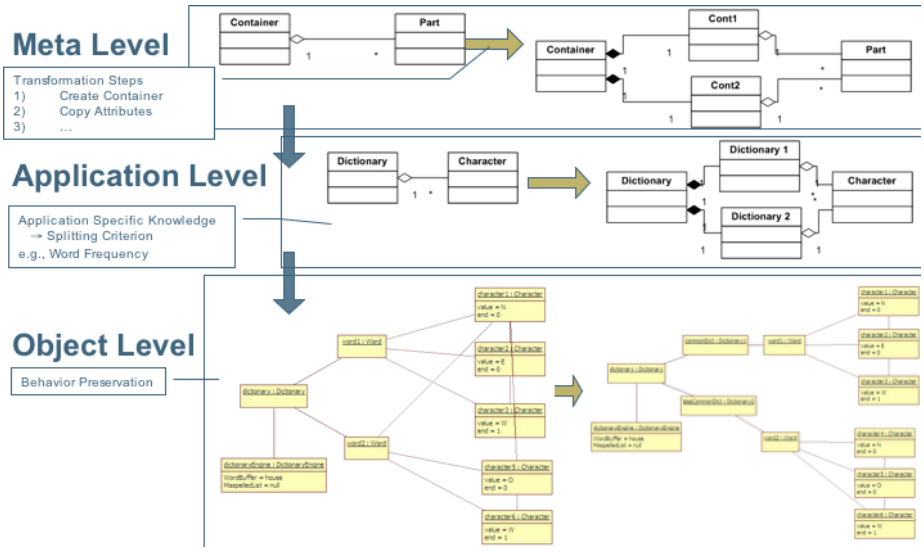

Figure 3.1: Different Levels of Abstraction

3.1 The Meta Level

The core of describing model transformations is to show, how a correct source model is transformed into a correct target model.

In order to define which models are correct (i.e., contain legal model elements in legal relationships), a metamodel is constructed. This metamodel defines a modeling language that is used to construct correct models. The metamodel itself is also written in a modeling language. For example, the UML metamodel is created in UML. The language that describes the metamodel is found in the Meta-Meta Level of abstraction (or M3 in the MDA context). In this work the language used to describe the metamodel is UML.

The following example shows the usage of metamodels. The UML metamodel contains the element *Argument* (Figure 3.2). Therefore *Argument* is part of the modeling language and can be used by a modeler to construct UML models. Figure 3.2 shows a part of the UML 1.4 metamodel[1], in which *Argument* is shown.

In this work the ADT metamodel is used to define ATDs. Every ADT model that is created must comply to this ADT metamodel.

The ADT metamodel was first introduced in [32]. Figure 3.3 shows a version created in [30].

[1] http://www.omg.org/technology/documents/formal/uml.htm

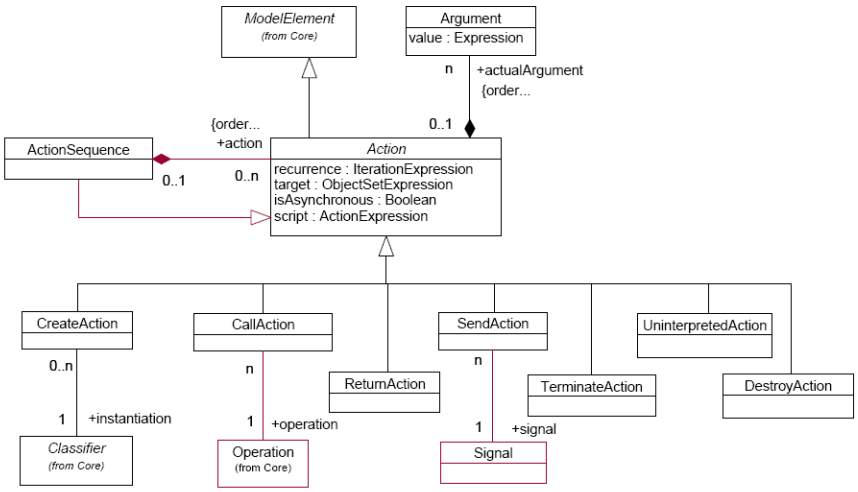

Figure 3.2: Excerpt of the UML 1.4 Metamodel

According to the metamodel an ADT consists of several related Concepts. Aggregation ([27], [5]) is used to model the hierarchy of the concepts in the ADT. Aggregation represents a whole-part relationship. In the metamodel the aggregate is called the *Container*. The *Part* concept is used to model the atomic objects in the ADT. Also ordinary associations between the concepts can exist. In the ADT metamodel, the association *C2C* indicates that *Concepts* can be associated at the class level. Both *Part* and *Container* have attributes (data) and operations.

Figure 3.3 shows how data is modeled and represented in ADT models. However, to fully specify what modeling elements can be used in an ADT model, further concepts need to be shown. (The extensions are partly adopted from the UML 1.4 metamodel.) Only the concepts required to understand the queries to the metamodel in Section 3.4 are shown here. They are *Association*, *Operation* and *Attribute*. It is not within the scope of this work to fully explain the ADT metamodel.

The queries in Section 3.4 to *Operation* and *Attribute* are straight forward and thanks to their naming easily understandable.

A little more complicated are the constructs involving *Associations*. Figure 3.4 shows the ADT metamodel together with the specifications for associations. An *Association* is composed of two *AssociationEnds*, where the *multiplicity* is specified. *Aggregation* in *AssociationEnd* defines whether *Association* is indeed an association, an aggregation or a composition.

To now transform a target model into a source model, their metamodels are consulted. (The distinction between source metamodel and target metamodel is made, because transformations between models constructed with the same language are only one specific case in the MDA framework.) A set of source model elements is related

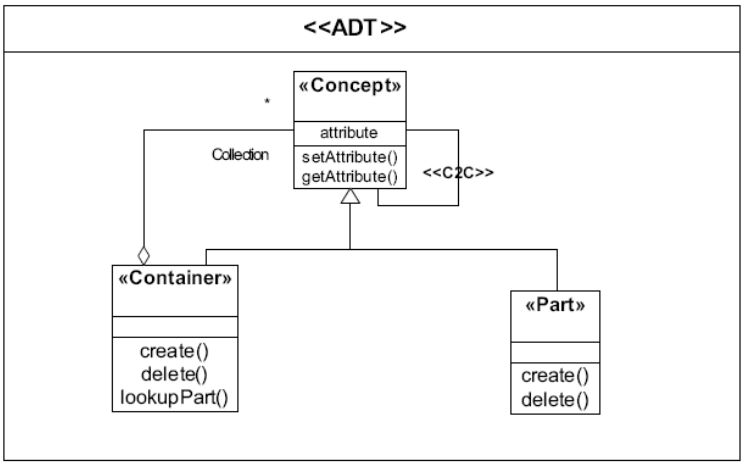

Figure 3.3: The ADT Metamodel expressed in UML

to a set of target model elements. Since a transformation can not always be applied and is not always correctly applied, conditions both on the source as well as the target model are defined. These conditions can be viewed as preconditions and postconditions that state under which circumstances a transformation can be applied and when the transformation was applied correctly. All this is captured in a transformation rule.

Since most transformations are complex and can be split into several atomic transformations, for each transformation a set of transformation rules is created. Thus, each transformation rule maps a set of source model elements to a set of target model elements and contains conditions, as described above.

[15] supplies a list of requirements needed to define a transformation rule. This list is given below. Further, they suggest a formal notation for writing transformation rules. The authors did not intend it as a proposal for a standard transformation language, but since the notation is close to OCL syntax and therefore easily understandable, it is adopted in this work.

"Any definition of a transformation rule should contain the following information:

- The source language reference.
- The target language reference.
- Optional transformation parameters, for example, constants used in the generation of the target.
- A set of named source language model elements (called S) from the source language metamodel.
- A set of named target language model elements (called T) from the target language metamodel.

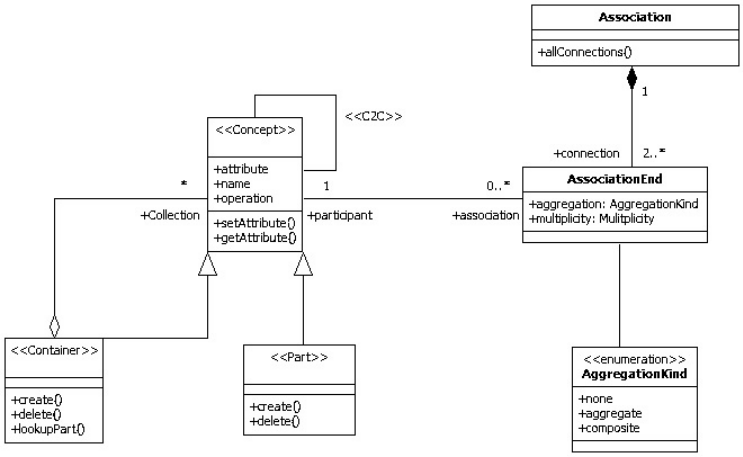

Figure 3.4: The ADT Metamodel extended with Association Specification

- A bidirectional indicator: a Boolean that states whether or not a source model may/can be (re)generated from the target.
- The source language condition: an invariant that states the conditions that must hold in the source model for this transformation rule to apply. The invariant may only be expressed on elements from set S.
- The target language condition: an invariant that states the conditions that must hold in the target model for this transformation rule to apply, or that needs to be generated when the target model is not yet present. The invariant may only be expressed on elements from set T.
- A set of mapping rules, where each rule maps some model elements in set S to model elements in set T."[15]

An example of a thoroughly defined transformation is provided in Section 3.4.

3.2 The Application Level

While the above explained Meta Level is used to describe the transformations, the Application Level (or M1, as it is called in the MDA framework) contains all information to make the transformations relevant and meaningful in the embedded systems context. The models at Application Level are the most relevant for the embedded systems designer.

The mere application of the transformations does not optimize the design of a software application. It is the implementation steered by application specific knowledge

that makes the transformations useful and relevant for embedded systems optimizations.

Also, transformations can not be applied on every model and on every application. The knowledge in the models combined with the designers domain knowledge, identifies and points towards transformation opportunities that lead to beneficial applications of the transformations.

And thirdly, concrete behavior and behavioral changes are best expressed with models that are able to capture the behavior of the application without relying too much on concrete instances. (It is shown in the next Section, how behavior preservation is expressed at the Object Level.)

Thus, at the Application Level all information regarding the concrete application, i.e., application/domain specific knowledge, transformation opportunities and behavioral changes, are expressed.

Since in Section 4.2.3 the application specific behavior (i.e., the splitting criterion) and the transformation opportunities for the Split Container transformation are already explained, in the following only the behavioral modeling is discussed.

Behavioral Modeling

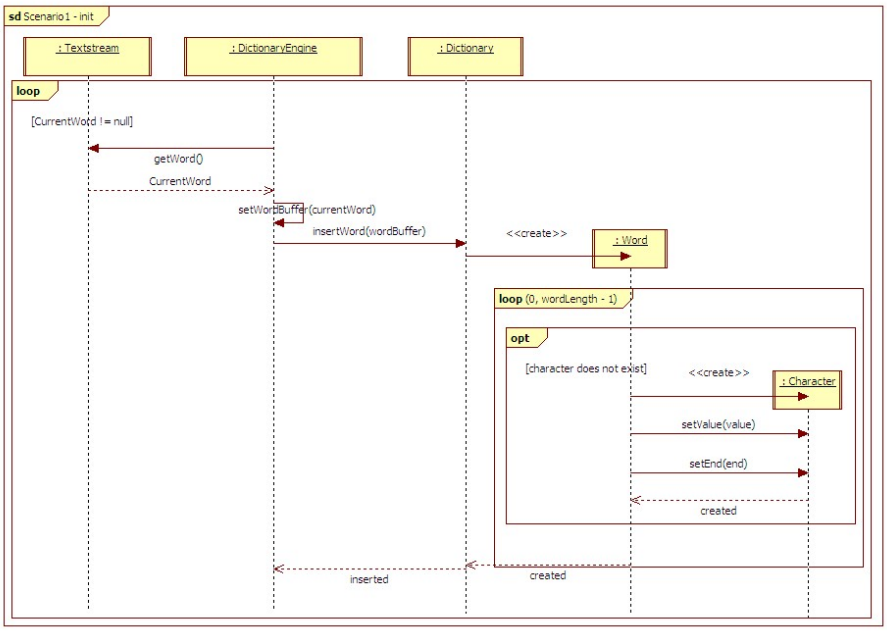

Figure 3.5: Sequence Diagram – Snapshot1

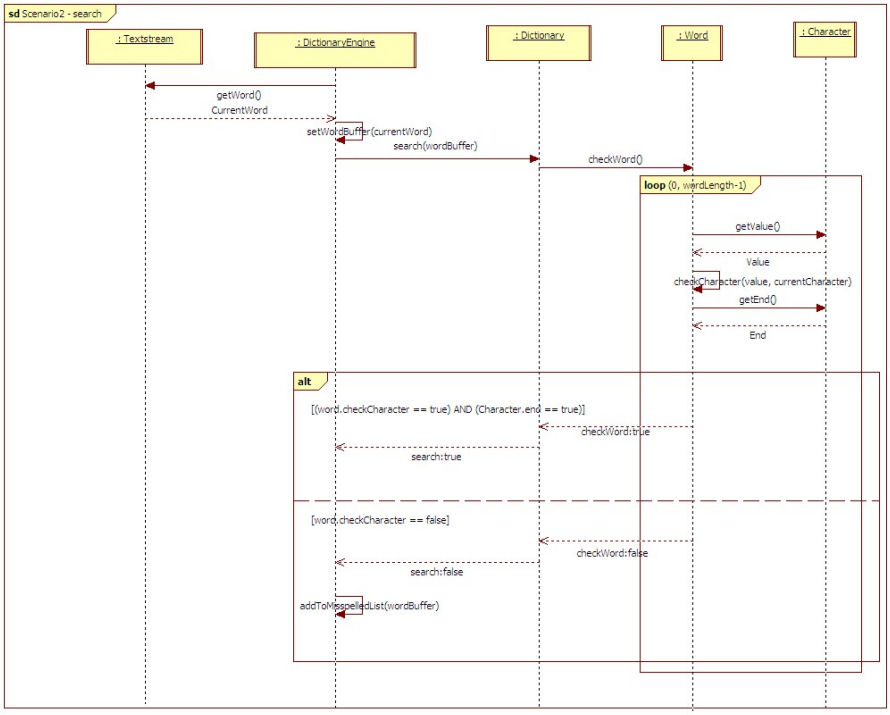

Figure 3.6: Sequence Diagram – Snapshot2

In the context of this work, behavioral diagrams are used to show the relevant behavior of an application, before and after a transformation. With correct behavioral modeling it is possible to follow changes in behavior and to later verify whether the behavior is preserved through the transformations. (Again, how behavior preservation is checked is shown in Section 3.3.)

Two sequence diagrams (Figure 3.5 and Figure 3.6) show the behavior of the spell checker application for two snapshots, Snapshot1 and Snapshot2. Snapshot1 consists of setting up the dictionary and Snapshot2 contains looking up a word in the dictionary. (The concept of snapshots is further explained in Section 3.3.)

Figure 3.7 and Figure 3.8 present the same sequence diagrams after the Split Container transformation was applied to the model. So Figure 3.7 shows the sequence diagram of Snapshot1, after the Split Container transformation was applied to it and Figure 3.8 shows the sequence diagram for Snapshot2, after the same transformation was applied to it.

The diagrams capture the relevant behavior of each snapshot and, comparing two diagrams for one snapshot, the concrete behavioral changes can be seen.

As an example, in Snapshot1 (Figure 3.5) a word is read from the *Textstream*, inserted into *wordBuffer*, *insertWord* is called on *Dictionary* and this triggers the creation

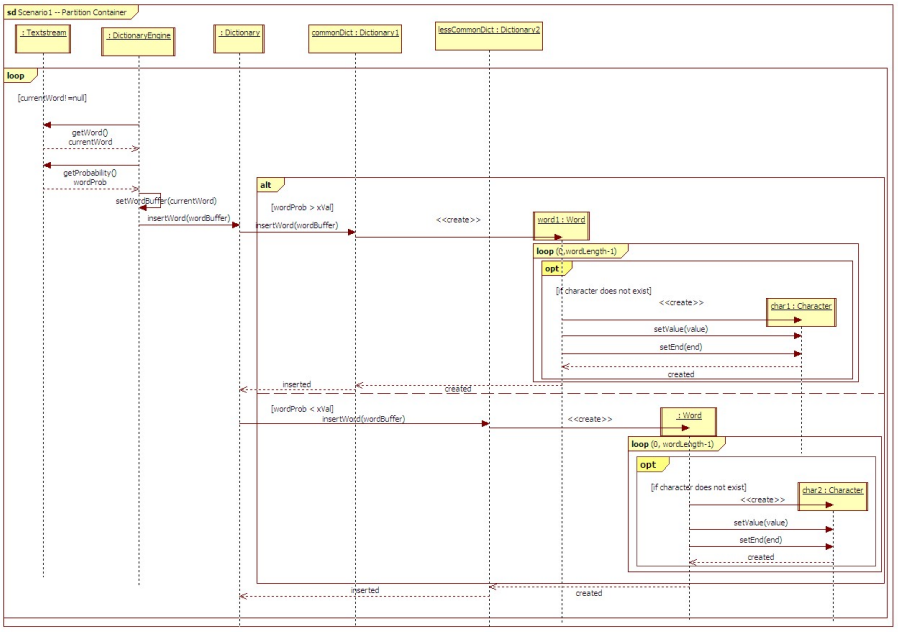

Figure 3.7: Sequence Diagram after Split Container Transformation – Snapshot1

of an instance of *Word* followed by the creation of the corresponding *Character* instances. If both *Characters* and *Word* are created correctly, `true` is returned, which triggers *insertWord* to also return `true` to *DictionaryEngine*.

After the Split Container transformation is applied, the insertion behavior is changed. Now, not only a word but also its probability is read from *Textstream*. Also, before a word is created, it is checked into which dictionary, *Dictionary1* or *Dictionary2*, it needs to be inserted.

In Snapshot2 the behavior is also changed through the Split Container transformation. In Figure 3.8 the changed part of the sequence diagram is depicted. The algorithm is changed in such that now each word is first looked up in *Dictionary1* and only if it is not found there, looked up in *Dictionary2*.

In the next section, it is shown that behavior is preserved during the model transformations.

3.3 The Object Level

Behavior preservation shown on the Split Container transformation

The concrete instances of the models from the Application Level exist at the Object Level (M0, in the MDA Framework). At this level the behavior preservation is shown.

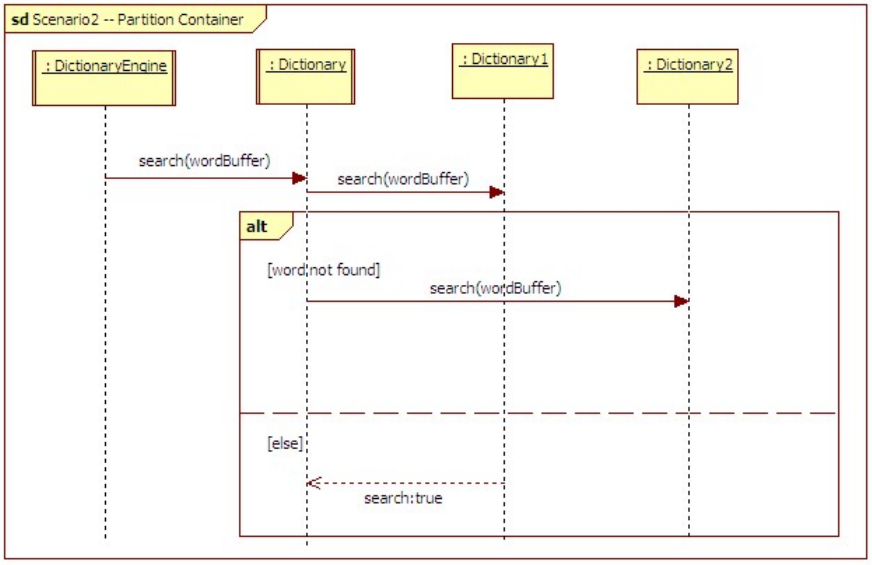

Figure 3.8: Sequence Diagram after Split Container Transformation – Snapshot2

In the following, a definition of behavior preservation from the Refactoring domain is presented and how a definition for this work is derived from it. Finally, this section concludes with an example.

Definition of behavior preservation:

> "The versions of a program before and after a refactoring must also produce *semantically equivalent references and operations*. Semantic equivalence is defined here as follows: let the external interface to the program be via the function *main*. If the function *main* is called twice (once before and once after a refactoring) with the same set of inputs, the resulting set of output values must be the same."[22], p. 28.

To apply the above definition in this work, it needs to be clarified what is considered to be input and output and how the execution (i.e., the run-time behavior) of the application is modeled.

To capture the run-time behavior of an application, snapshots are used. A snapshot consists of the run-time behavior of an application up to a specific point, i.e., it captures an instance of data at an instance of time.

Here, snapshots are used to model the applications execution, this is, to capture concrete objects with their specific values for a certain execution path. Figure 3.14 shows an example of a modeled snapshot.

According to the behavior preservation definition from above, it can be said that,

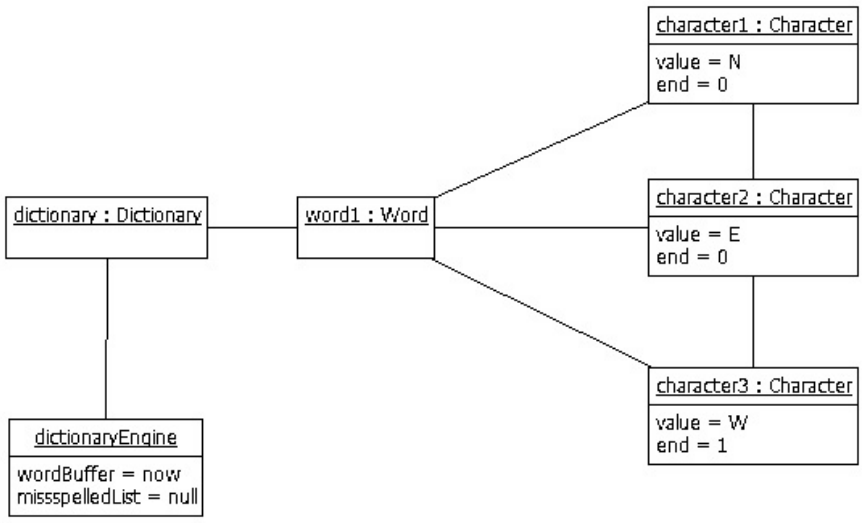

Figure 3.9: Object Diagram Snapshot1, before the transformation

behavior is preserved if two versions of one snapshot, one before the transformation and one after the transformation, produce the same output for the same input.

Thus, comparing the two versions of a snapshot (before and after the transformation took place) preserved behavior can be shown. If the same relevant data is created and relevant methods return the same results, the behavior is the same at modeling level.[2]

Here 'relevant' means that only a subset of data and behavior is considered. I.e., behavior preservation is only shown for a certain focused-on functionality. In every transformation changes occur most likely in various parts of the application and thus possibly change parts of the behavior. To give an insight in the meaning of 'relevant behavior', consider the following example. In the Split Container transformation, two new dictionaries are created and the words are distributed among those dictionaries. To look up a word, it is first searched in one dictionary, then in the other. This is a clear change in behavior. However, the focused-on behavior is searching a word. The *search* method should return in both cases the same result: whether a word is found in the dictionary or not. As long as this behavior is not changed and the same word (i.e., relevant data) is found in the dictionary before and after the Split Container transformation is applied, the other changes in behavior are ignored.

Thus identifying relevant data and relevant functionality is as important (and as difficult) as identifying relevant snapshots.

In the following the application of this definition is shown on a more concrete example, the Split Container transformation.

[2]This, of course, can not make any final judgment about implementations and their execution, since the translation from model to code is not specified in this work.

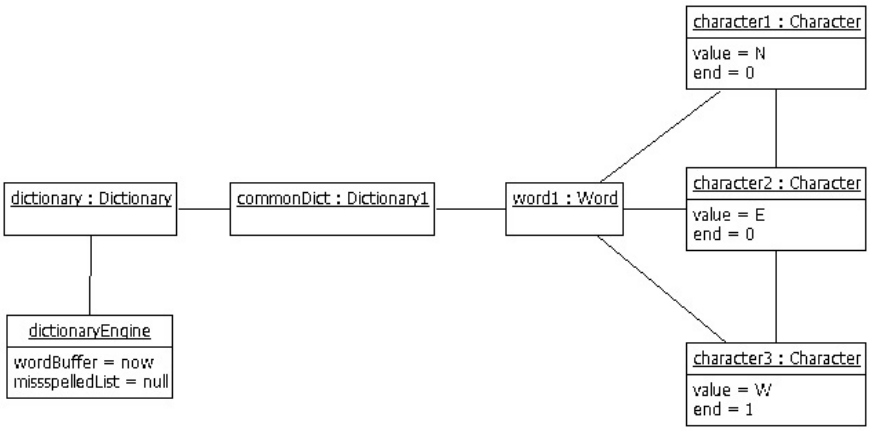

Figure 3.10: Object Diagram, Snapshot1, after the transformation

For the spell checker application two snapshots are identified.

As mentioned before, Snapshot1 describes setting up the dictionary. In the following example the dictionary is initially empty and the word 'new' is inserted. The end of the snapshot is reached, when the dictionary engine receives the feedback that the word was successfully created. (Figure 3.5)

Snapshot2 contains the lookup of a word in the dictionary. Here the dictionary contains the words 'new' and 'now' and the word 'now' is checked in the dictionary. The snapshot ends when the lookup method returns `true`. (Figure 3.6)

For both snapshots two versions are considered, one, the snapshot before the transformation is performed and two, after the transformation is performed. Figures 3.9, 3.10, 3.11 and 3.12 show object diagrams for Snapshot1 and Snapshot2 before and after the transformation. In the object diagrams the changes to the application's data are depicted. Through the Split Container transformation, more character objects are created, due to the use of two dictionaries instead of one. In the following, it is shown, how the behavior is preserved in Snapshot1, despite the changes of the data values.

Figures 3.13 and 3.14 show the sequence diagrams for Snapshot1, before and after the Split Container transformation is applied.

To verify that the behavior is preserved, the two sequence diagrams describing one snapshot are queried.

First an input of interest needs to be defined, which is then compared.

In Snapshot1, a word is added into the dictionary. Thus, the word to be added, i.e., the word stored in *wordBuffer*, is the input. If in both versions the value of *wordBuffer* is the same, both versions are assumed to have the same input. In Figure 3.13 and Figure 3.14 the value of *wordBuffer* is 'new', thus they have the same input.

If, for this input, the output is the same in both versions, the relevant behavior is preserved.

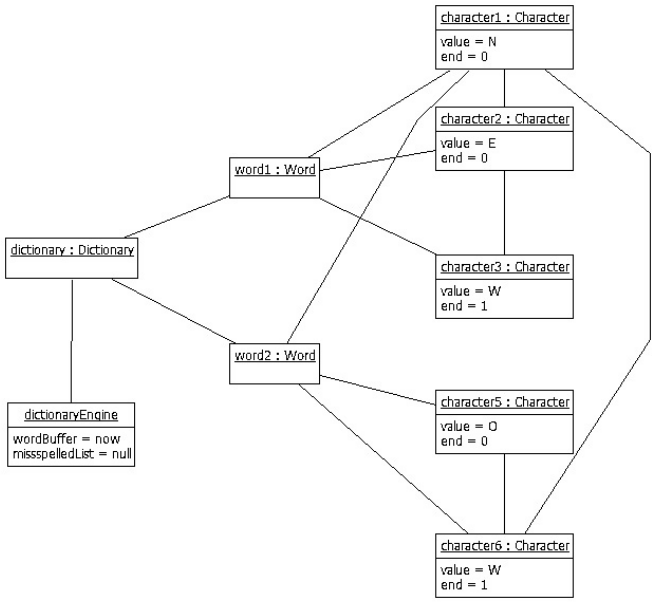

Figure 3.11: Object Diagram, Snapshot2, before the transformation

Here again, the relevant output needs to be defined. Whenever objects are created via a message call, a return message is send back to confirm whether the object was created successfully. For Snapshot1 the main behavior is the insertion of a word in the dictionary. Therefore the feedback on whether a word was created is considered the main (i.e., desired) output. As shown in Figure 3.5, at the end of the message sequence *DictionaryEngine* receives the Boolean return value. (This was triggered through *Dictionary* indicating that the word is created, after the characters are created or exist already.) In both Figure 3.13 and Figure 3.14 *DictionaryEngine* receives a `true` message. Therefore the output is the same.

According to this reasoning, it is verified that the behavior is preserved during the application of the Split Container transformation.

To verify the behavior preservation for Snapshot2 the same methodology is followed. There the input is again the current word in the word buffer. In Scenario2 a word is checked in the dictionary. As relevant output the return value of the search function is chosen.

3.4 Transformation Definition – Example

In the following subsection, a transformation definition on the Split Container transformation is presented. The Split Container transformation consists of several trans-

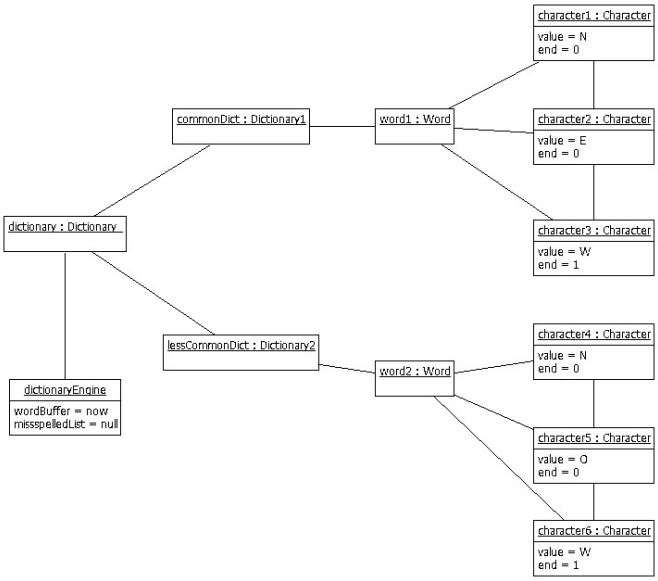

Figure 3.12: Object Diagram, Snapshot2, after the transformation

formation steps, and only the sequence of these steps fully defines the transformation. (Figure 2.5 show the inital and transformed model.)

Transformations Steps (or transformation rules as in [15]):

- create two new classes
- copy methods and attributes from the source class
- create composition relationship between each new class and the source class
- connect new classes with prior relationship type to part class
- remove attributes from source class
- delete relationship between old container and class

Short Introduction to OCL

Each step is defined as a transformation rule following the notation in [15]. OCL is used to express the invariants, i.e., the source language conditions and the target language conditions. To verify the correctness of the OCL syntax the OCLE tool is used.[3]

Before proceeding, the language constructs of OCL used in this work are shortly introduced.

[3]OCLE 2.0 – Object Constraint Language Environment, lci.cs.ubbcluj.ro/ocle

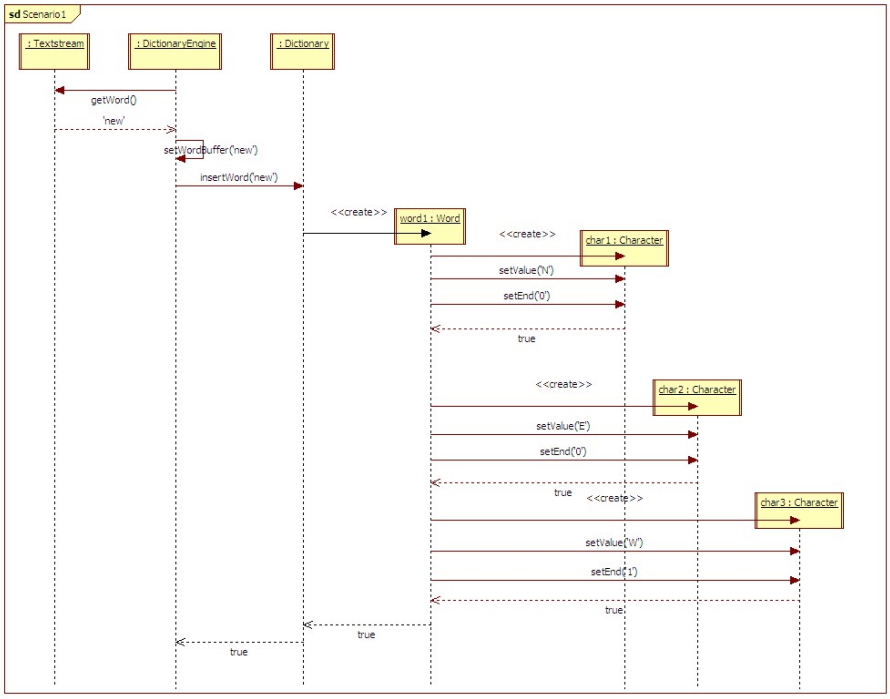

Figure 3.13: Detailed Sequence Diagram before Split Container Transformation – Snapshot1

OCL ([21], [34], [24]) is a language addition to UML that allows the modeler to express constraints and queries on models. Although OCL is an easy to read language, in the following the very broad basics are presented.

Basic Types and Collection Types

OCL has the following basic types:

- Boolean
- Integer
- Real
- String

For collections it has the types

- Set – A *Set* contains a collection of valid OCL types. In a *Set* each element **exists only once** and the elements are **not ordered**.

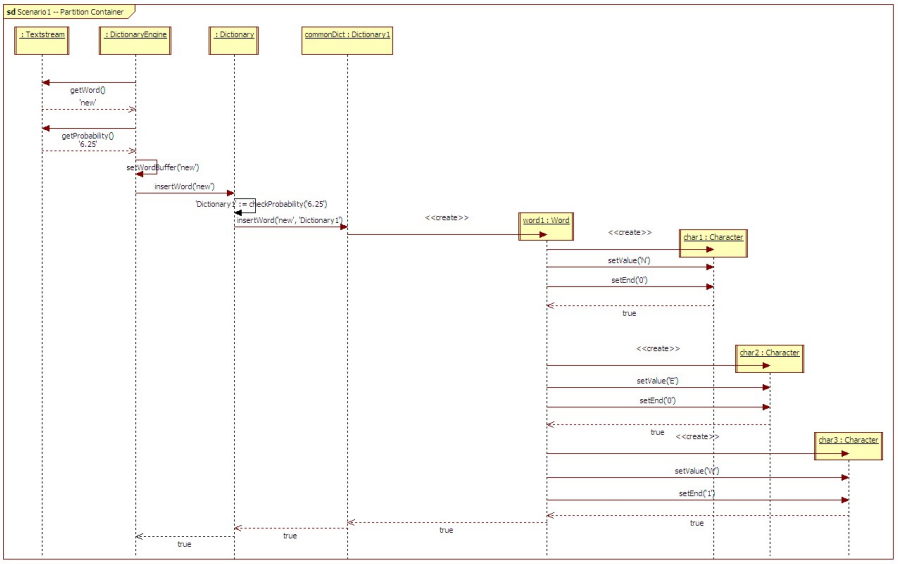

Figure 3.14: Detailed Sequence Diagram after Split Container Transformation – Snapshot1

- OrderedSet – An *OrderedSet* contains a collection of valid OCL types. In an *OrderedSet* each element only **exists once** and the elements are **ordered**.
- Bag – A *Bag* contains a collection of valid OCL types. In a *Bag* each element can exist **more than once** and the elements are **not ordered**.
- Sequence – A *Sequence* contains a collection of valid OCL types. In a *Sequence* each element can exist **more than once** and the elements are **ordered**.

are used.

Operations

In OCL a couple of operations that are applicable **on any OCL instance**, i.e., any modeling element, are defined.

The only one used in this work is `allInstances()`. It returns a set of all instances of the modeling element.

`type::allInstances():Set(type)`

A set of operations on **all collection types** is defined in OCL. The following can be found in this work.

- `excludes(object)` – Returns `true` if *object* is not part of the collection.

- `Collection.excludesAll(collection)` – Returns `true` if none of the elements of *collection* belongs to *Collection*, i.e., the collection the operation is applied on.
- `includes(object)` – Returns `true` if *object* is part of the *collection*.
- `isEmpty()` – Returns `true` if the *collection* is empty.

Of the predefined **loop operations** that iterate over collections the following are used:

- `forAll(expression)` – Returns `true` if all elements of the collection are `true` for the *expression*.
- `exists(expression)` – Returns `true` if at least one element of the collection is `true` for the *expression*.

Detailed Execution

Create two new containers

1. source language reference: ADT metamodel
2. target language reference: ADT metamodel
3. transformation parameters:
```
cont1Name :  String = container1.name
cont2Name :  String = container2.name
allContainers :  Bag (String) = Container
.allInstances().name
```
4. set of named source language model elements: - - *none*
5. set of named target language model elements:
```
container1 :  ADT::Container
container2 :  ADT::Container
```
6. (bidirectional indicator): `unidirectional`
7. source language conditions:
   names are not already taken
```
(allContainers->excludes(cont1Name))
and (allContainers->excludes(cont1Name))
```
8. target language conditions:
   container1 and *container2* were created
```
(container1 <> "") and (container2 <> "")
```
9. mapping rules: –

In the notation, used in [15] and adopted from here onwards, this translates to:

Transformation `CreateNewContainers(ADT, ADT) {`
params
```
cont1Name :  String = container1.name;
cont2Name :  String = container2.name;
allContainers :  Bag (String) = Container
   .allInstances().name;
```
source
- - none
target
```
container1 :  ADT::Container;
container2 :  ADT::Container;
```
source condition
```
(allContainers->excludes(cont1Name))
and (allContainers->excludes(cont1Name));
```
target condition
```
(container1 <> "") and (container2 <> "");
```
unidirectional;
mapping
– none
}

Copy methods and attributes from source container to newly created containers

Transformation `CopyAttributesAndOperations(ADT, ADT) {`
params
– none
source
```
sourceContainer :  ADT::Container;
sourceAttribute :  ADT::Container.attribute;
sourceOperation :  ADT::Container.operation;
sourceCont1 :  ADT::Container;
sourceCont2 :  ADT::Container;
```
target
```
targetContainer :  ADT::Container;
targetAttribute :  ADT::Container.attribute;
targetOperation :  ADT::Container.operation;

targetCont1 :  ADT::Container;
targetCont1Attribute :  ADT::Container.attribute;
targetCont1Operation :  ADT::Container.operation;

targetCont2 :  ADT::Container;
targetCont2Attribute :  ADT::Container.attribute;
targetCont2Operation :  ADT::Container.operation;
```

source condition
– *none*
target condition
– *methods and attributes exist in both new classes*

```
(targetCont1Operation.name = targetOperation.name)
and
(targetCont2Operation.name = targetOperation.name)
and
(targetCont1Operation.type = targetOperation.type)
and
(targetCont2Operation.type = targetOperation.type)
and
(targetCont1Operation.parameters
  = targetOperation.parameters)
and
(targetCont2Operation.parameters
  = targetOperation.parameters)
and
(targetCont1Operation.type = targetOperation.type)
and
(targetCont2Operation.type = targetOperation.type)
and
(targetCont1Operation.owner = targetOperation.owner)
and
(targetCont2Operation.owner = targetOperation.owner)
and
(targetCont1Attribute.type = targetAttribute.type)
and
(targetCont2Attribute.type = targetAttribute.type)
and
(targetCont1Attribute.name = targetAttribute.name)
and
(targetCont2Attribute.name = targetAttribute.name)
and
(targetCont1Attribute.owner = targetAttribute.owner)
and
(targetCont2Attribute.owner = targetAttribute.owner);
```

unidirectional;
mapping

```
sourceContainer <-> targetContainer;
sourceAttribute <-> targetAttribute;
sourceOperation <-> targetOperation;
sourceCont1 <-> targetCont1;
sourceCont2 <-> targetCont2;
```

}

Create composition relationship between each new container and the source container

```
Transformation CreateComposition(ADT, ADT) {
    params
      connectedContainerNames :  Bag (String)
        = Bag{targetContainer.name, targetCont1.name};
    source
      sourceContainer :  ADT::Container;
      sourceCont1 :  ADT::Container;
      sourceCont2 :  ADT::Container;
    target
      targetContainer :  ADT::Container;
      targetCont1 :  ADT::Container;
      targetCont2 :  ADT::Container;
      composition1 :  ADT::Association;
      composition2 :  ADT::Association;
    source condition
      - - none
    target condition
      - - between containers created
      composition1.connection.participant.name
        ->includes(connectedContainerNames)
      and
      - - is composition
      composition1.allConnections
        ->exists(aggregation = AggregationKind::composite)
      composition2.allConnections
        ->exists(aggregation = AggregationKind::composite)
      and
      - - one-to-one multiplicity
      composition1.allConnections
        -> forAll(multiplicity = 1)
      composition1.allConnections
        ->forAll(multiplicity = 1);
    unidirectional;
    mapping
      sourceContainer <-> targetContainer;
      sourceCont1 <-> targetCont1;
      sourceCont2 <-> targetCont2;
}
```

connect new classes with prior relationship type to part class

```
Transformation connectNewClasses(ADT, ADT) {
    params
```

```
      connectedContainerNames :  Bag (String) =
        Bag{targetContainer.name, targetCont1.name};
      priorAggregateKind:AggregateKind
        = AggregationKind::composite;
```

source

```
      sourceContainer :  ADT::Container;
      sourceCont1 :  ADT::Container;
      sourceCont2 :  ADT::Container;
```

target

```
      targetContainer :  ADT::Container;
      targetCont1 :  ADT::Container;
      targetCont2 :  ADT::Container;
      composition1 :  ADT::Association;
      composition2 :  ADT::Association;
```

source condition

- - none

target condition

- - same type as before

```
      composition1.allConnections()
        ->exists(aggregation = priorAggregateKind);
```

- - between containers created

```
      composition1.connection.participant.name
        ->includes(connectedContainerNames);
```

- - is composition

```
      composition1.allConnections
        ->exists(aggregation = AggregationKind::composite);
      composition2.allConnections
        ->exists(aggregation = AggregationKind::composite);
```

- - one-to-one multiplicity

```
      composition1.allConnections
        ->forAll(multiplicity = 1);
      composition1.allConnections
        ->forAll(multiplicity = 1);
```

unidirectional;

mapping

```
      sourceContainer <-> targetContainer;
      sourceCont1 <-> targetCont1;
      sourceCont2 <-> targetCont2;
}
```

remove attributes from source container

Transformation `RemoveAttribute(ADT, ADT) {`

params

- - none

source

```
      attribute :  ADT::Container.Atttribute;
      attributeGetter :  ADT::Container.operation;
      attributeSetter :  ADT::Container.operation;
      sourceContainer :  ADT::Container;
   target
      targetContainer :  ADT::Container;
   source condition
      - - attribute is unreferenced
      attributeGetter.parameter->select(kind
        = ParameterDirectionKind ::  return)
        ->excludes(attribute);
      attributeSetter.parameter->select(kind
        = ParameterDirectionKind ::  in)
        ->excludes(attribute);
   target condition
      - - attribute was deleted
      targetContainer.allAttributes->isEmpty;
   unidirectional;
   mapping
      - - none
}
```

delete relationship between old container and class

```
Transformation DeleteRelationship(ADT, ADT) {
   params
      sourceNames :  Bag (String) = Bag{
        sourceContainer.name, sourcePart.name};
      targetNames :  Bag (String) = Bag{
        targetContainer.name, targetPart.name};
   source
      sourceAggregation:  ADT::Association;
      sourceContainer:  ADT::Container;
      sourcePart:  ADT::Part;
   target
      targetContainer :  ADT::Container;
      targetPart :  ADT::Part;
   source condition
      - - Aggregation between part and container exists
      (sourceAggregation <> "") and
      (sourceAggregation.connection.participant.name
        ->includes(sourceNames));
   target condition
      - - Aggregation was deleted
      ADT::Association.allInstances()
        ->select(connection.participant.name
```

```
            ->excludesAll(targetNames));
    unidirectional;
    mapping
       sourceContainer <-> targetContainer;
       sourcePart <-> targetPart;
}
```

Chapter 4

Profiling Results

In this chapter the results of implementing and profiling various UML model transformations are presented. The results show reduction of memory footprint, data accesses and overall energy consumption. In the first section, implementation details concerning the application, its various dynamic inputs, the chosen data structures and the C code implementation are shown. In the remaining chapter, the results are presented in data tables with explanatory notes, highlighting the specific results for each transformation.

4.1 Implementation Details

4.1.1 Application

The driver application, used to show the impact of the proposed transformations, is the spell checker discussed earlier in this work. The spell checker has the following basic functionality: To spell-check a text, each word is looked up in the internal dictionary. If the word is not found in this dictionary, it is added to a list of misspelled words. The dictionary is set up in the beginning by adding words to it.

4.1.1.1 Trie ADT

The spell checker's dictionary is implemented with a Trie ADT [11]. There are several concrete data structure implementations (cf. [4]) for the Trie ADT. The goal is to use two widely known implementations that each represent one extreme in the concrete data structure design space, i.e., the extremes that are Pareto-optimal in terms of memory footprint and data accesses. A Pareto-optimal point represents an optimal solution in one trade-off direction when the other direction is fixed. ([12]) Thus, an array based version, as well as a linked-list based version are chosen, namely an array-structured Trie and a list-structured Trie are implemented. (Both are explained in [2].)

The implementation that has little data accesses and a large memory footprint is the array-structured Trie. (Depicted in Figure 4.1.) The array-structured Trie uses static arrays. That means that each array contains 26 cells, one for each letter of the alphabet. The advantage is, that this allows direct access to the values in the array. On the other

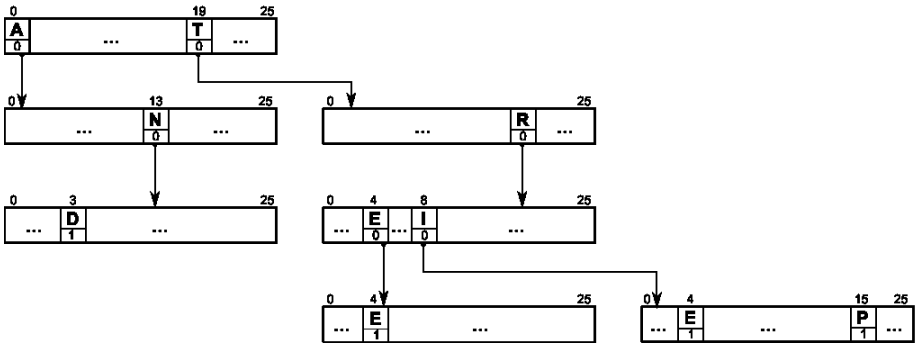

Figure 4.1: Array-Structured Trie for the words ‘and’, ‘tree’, ‘trie’, ‘trip’

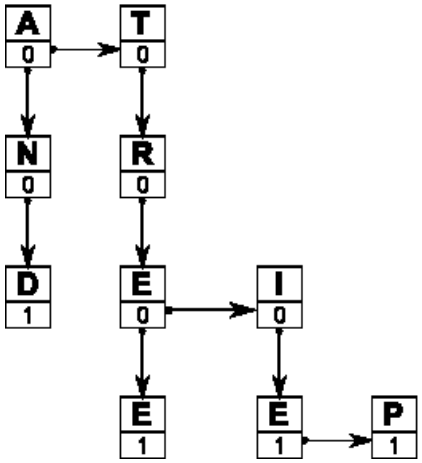

Figure 4.2: List-Structured Trie for the words ‘and’, ‘tree’, ‘trie’, ‘trip’

hand however, this increases the memory footprint a lot. Especially for long words or small dictionaries, the arrays exhibit a lot of sparseness.

The implementation that has a smaller memory footprint but uses more data accesses is the list-structured Trie (Figure 4.2). The large number of accesses is due to the fact that here linked lists are created. These lists need to be traversed to lookup a word in the dictionary, which results in additional accesses. On the other hand, the list-structured Trie is space efficient compared to the array-structured Trie, each node only contains its value, the end flag and two pointers. Further, nodes are only created if the character exists in the dictionary. This avoids the sparseness found in the arrays of the array-structured Trie.

4.1.2 Input Characterization

4.1.2.1 Dictionary

The dictionary of the spell checker is set up with a word list (provided by [14]) consisting of 57 046 English words.[1] The list also provides the relative occurrence frequency of each word in the English language. The frequency information is ignored in the initial implementation, but is used for later transformations. It is indicated where this is the case.

[Since the original large dictionary of approximately 57 000 words allocates a large amount of memory (i.e., about 32MB) for the array-structured Trie implementation, all transformations are also profiled with a reduced dictionary of 20 833 words. Where it is used, it is referred to as small or reduced dictionary.]

4.1.2.2 Text

Text	spiegel	independent	theindependent	spiegelLong	economist	economistShort	economistLong	bbc
Length	1 107	406	753	7 764	1 078	591	3 001	303
Erroneous words	132	26	34	1 072	172	41	256	36
Error rate [%]	11,92	6,4	4,52	13,81	15,95	6,94	8,53	11,88
Duplicate words	55	0	6	577	75	2	57	12
Duplication Rate [%]	41.6	0	17.6	53.5	43.6	4.9	22.3	3 001

Table 4.1: Characterization of the input texts

The application is profiled with eight different texts taken from various newspapers and news magazines as input. Thus the texts have one genre in common. This yields texts with similar length and a similar vocabulary. Further, those texts commonly do not contain wrongly spelled words, words detected as misspelled are mostly names of persons and places. This yields similar behavior patterns, i.e., for eight different texts

[1] excerpt from the READ ME: "The word list is primarily intended to be useful for checking spelling. [..] Principal omissions:
- words requiring a capital letter
- abbreviations
- slang
Colloquialisms and archaisms are generally excluded. Contractions are excluded. A rare word similar to a common word may be excluded. Both -ise and -ize spellings are included. The character set is: lowercase letters, hyphen, apostrophe. Words which can be spelled with accents occur here in their plain form."

Text	spiegel	independent	theindependent	spiegelLong	economist	economistShort	economistLong	bbc
Length	1 107	406	753	7 764	1 078	591	3 001	303
Erroneous words	152	34	40	1 239	209	58	333	41
Error rate [%]	13.7	8.4	5.3	15.92	19.4	9.8	11	13.5
Duplicate words	59	1	6	610	78	3	73	12
Duplication Rate [%]	38.8	2.9	15	49.2	37.3	5.2	21.9	29.3

Table 4.2: Characterization of input text behavior with the small dictionary

an error rate between 4,92% and 15,95% and an error re-occurrence rate up to 53,5% is noted for the original dictionary. The error re-occurrence of 53,5% describes that up to every second misspelled word is a duplicate. Table 4.1 shows the concrete values for the eight input texts looked up in the original large dictionary. Table 4.2 shows the results for the same texts looked up in the reduced dictionary.

4.1.3 Implementation in C

Initially, the spell checker application exists only in a UML model. (Figure 4.3 shows again the initial UML model of the spell checker application.) To profile the application, these models are manually transformed into C code. Since UML models the application at a higher level of abstraction, some specific decisions are made to map the model to C code.

A class consists of attributes and behavior (methods). The general methodology is to translate all attributes of one class into one type, while the behavior of the class is encapsulated into header files.

Specifically this means, the class *Character* forms a type consisting of *Character's* attributes. Note that the association relationship *next* is translated into a *next* pointer and the *down* pointer is generated by consulting the weak aggregation relationship with *Word*.

The weak aggregation relationship between *Word* and *Character* expresses the notion of letters (or characters) in a word. A *Word* is a container holding several *Characters*. Further, the weak aggregation, as opposed to a strong aggregation or composition, expresses that *Characters* can be shared among instances of *Word* – plainly, characters are re-used for different words (see Figure 4.2). The *down* pointer is used to express this relationship. Type for *Character*:

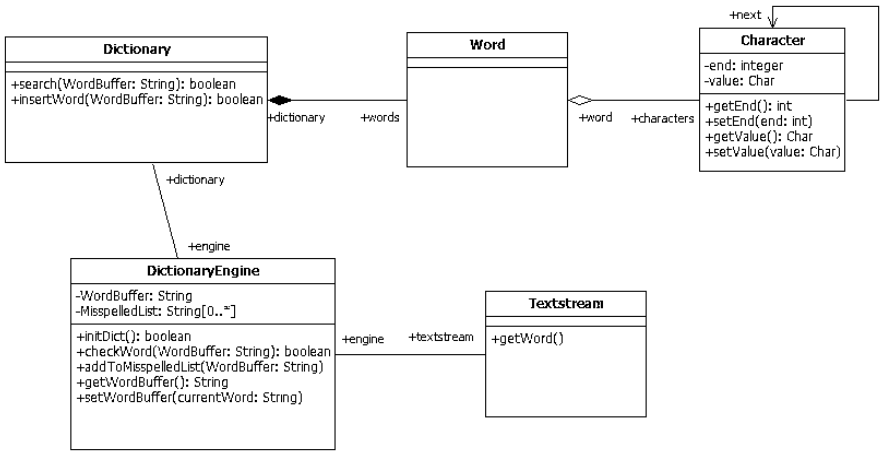

Figure 4.3: UML Class Diagram of the Spell Checker

```
struct character {
 char value;
 int end;
 character * next;
 character * down;
};
```

Word is a purely conceptual class in the sense that it describes the notion of a word consisting of a sequence of characters. It holds neither methods nor data and does not need to be instantiated.

The class *Dictionary* contains the methods that operate on the dictionary's data, *search* and *setup*. The strong aggregation relationship from *Dictionary* to *Word* describes that a dictionary consists of words. *Search* and *setup* are declared in a header file and are called by *DictionaryEngine.*

The class *DictionaryEngine* holds the main functionality of the application. From here the methods *setup* and *search* are called. It also contains the *misspelledWords* list and the *wordBuffer*. *DictionaryEngine* is represented by the *main* function. Also a type *DictionaryEngine* exists that holds the two arrays *wordBuffer* and *misspelledList*.

4.2 Profiling

4.2.1 Experimental Setup

The profiling is done with the profiling libraries described in [25]. Each read or write to a variable is counted as one data access. In this work data accesses are used instead

of memory accesses. This conveys that here only the algorithmic accesses to the data are counted. Dynamic allocation (malloc) overhead, and such, is not considered in this. The memory footprint of the data structures is determined using the allocation information provided by the profiling library. Energy is calculated based on the energy model provided by [23] for embedded SRAMs. Energy calculations are considered for an SRAM size of up to 16 MB, because the model in [23] does not provide energy figures for larger SRAM sizes. Therefore, where the memory footprint exceeds 16 MB no energy values are given and the energy values for the reduced dictionary are pointed out instead.

The application's execution is decomposed into two scenarios. Scenario1 consists of setting up the dictionary. This means that each word of the word list is inserted in the Trie. Scenario2 describes the lookup of the input text in the dictionary. Later, accesses, memory footprint and energy figures are given for the different scenarios in addition to the total numbers.

Results for the total memory footprint of the application, as well as the memory footprint of the Trie and the memory footprint of the used buffers are given. Buffers in this case are the dynamically allocated *misspelledList* and the *wordBuffer*. The memory footprint of *misspelledList* represents the memory footprint at the end of the application's execution with all the misspelled words. For the memory footprint of *wordBuffer* the maximum size of a *wordBuffer*, occurring during the execution is taken.

In the following, tables with the results of profiling different transformations are provided for data accesses, memory footprint and energy consumption.

4.2.2 Initial Implementation

Table 4.3 and Table 4.5 show the results for the initial list-structured Trie implementation. The first table shows the results for the original dictionary while the second table shows the results for the reduced dictionary. Table 4.4 and Table 4.6 hold the results for the initial array-structured Trie implementation.

Here it is interesting to compare the array-structured with the list-structured solution. In Section 4.1.1 it is predicted that an array-structured implementation allocates more memory but uses less accesses than a list-structured implementation. This assumption is validated here.

Indeed, the list-structured implementation needs noticeably more data accesses than the array-structured implementation (approximately factor of two). The array-structured solution on the other hand allocates memory by a factor of 10 more. Energy is not compared for the solutions based on the large original dictionary, since the energy model does not consider SRAMs larger than 16 MB and the 35 MB of memory that the array-structured implementation allocates (Table 4.4) clearly exceed this. However, with a reduced dictionary, energy comparisons show that a list-structured solution is more energy efficient. This is due to the data access and storage pattern described above. Figure 4.4 shows the trade-off points between memory footprint and data accesses for the list-structured, as well as the array-structured Trie implementation. For this, the example of the text *spiegel* being looked up in the small dictionary is used.

Text	spiegel	independent	theindependent	spiegelLong
Accesses – total	8 094 697	8 021 345	8 054 768	8 793 615
Scen1 (setup)	7 979 343	<–	<–	<–
Scen2 (search)	115 354	42 002	75 425	814 272
Alloc. Memory – total[B]	2 379 320	2 376 776	2 376 968	2 401 880
Buffers [B]	3 192	648	840	25 752
Trie [B]	2 376 128	<–	<–	<–
Energy – total [nJ]	38 409 337	38 061 282	38 219 874	41 725 703
Energy-Scen1 [nJ]	37 861 982	<–	<–	<–
Energy-Scen2 [nJ]	547 355	199 299	357 892	3 863 721

Text	economist	economistShort	economistLong	bbc
Accesses - total	8 088 474	8 039 767	8 290 316	8 010 836
Scen1 (setup)	7 979 343	<–	<–	<–
Scen2 (search)	109 131	60 424	310 972	31 493
Alloc. Memory – total[B]	2 380 280	2 377 136	2 382 296	2 377 016
Buffers [B]	4 152	1 008	6 168	888
Trie [B]	2 376 128	<–	<–	<–
Energy – total [nJ]	38 379 809	38 148 694	39 337 549	38 011 417
Energy-Scen1 [nJ]	37 861 982	<–	<–	<–
Energy-Scen2 [nJ]	517 826	286 712	1 475 562	149 434

Table 4.3: Initial List-structured Trie Implementation – Original Dictionary

Text	spiegel	independent	theindependent	spiegelLong
Accesses – total	3 106 426	3 088 964	3 097 495	3 279 238
Scenario1	3 078 910	<–	<–	<–
Scenario2	27 516	10 054	18 585	200 328
Alloc. Memory				
total [B]	34 647 048	34 644 504	34 644 696	34 669 608
Trie [B]	34 643 856	<–	<–	<–
Buffers [B]	3 192	648	840	25 752
Energy – total [nJ]	—	—	—	—

Text	economist	economistShort	economistLong	bbc
Accesses – total	3 106 380	3 093 829	3 156 237	3 086 385
Scenario1	3 078 910	<–	<–	<–
Scenario2	27 470	14 919	77 327	7 475
Alloc. Memory				
total [B]	34 648 008	34 644 864	34 650 024	34 644 744
Trie [B]	34 643 856	<–	<–	<–
Buffers [B]	4 152	1 008	6 168	888
Energy – total [nJ]	—	—	—	—

Table 4.4: Initial Array-structured Trie Implementation – Original Dictionary

Text	spiegel	independent	theIndependent	spiegelLong
Accesses – total	2 712 209	2 643 649	2 675 248	3 364 289
Acc – Scen1	2 604 432	<–	<–	2 604 432
Acc – Scen2	107 774	39 214	70 813	759 854
Alloc Mem – total [B]	862 408	859 576	859 720	888 496
Alloc Mem – Trie [B]	858 736	<–	<–	<–
Alloc Mem – Buffers [B]	3 672	840	984	29 760
Energy – total [nJ]	5 136 924	5 007 071	5 066 920	6 371 963

Text	economist	economistShort	economistLong	bbc
Accesses – total	2 705 646	2 660 865	2 894 628	2 633 928
Acc – Scen1	2 604 432	2 604 432	2 604 432	2 604 432
Acc – Scen2	101 211	56 430	290 193	29 493
Alloc Mem – total [B]	863 776	860 152	866 752	859 744
Alloc Mem – Trie [B]	858 736	<–	<–	858 736
Alloc Mem – Buffer [B]	5 040	1 416	8 016	1 008
Energy – total [nJ]	5 124 493	5 039 678	5 482 425	4 988 660

Table 4.5: Initial List-structured Trie Implementation – Small Dictionary

Text	spiegel	independent	theIndependent	spiegelLong
Accesses – total	1 052 164	1 034 721	1 043 183	1 224 932
Acc – Scen1	1 024 608	<–	<–	<–
Acc – Scen2	27 556	10 113	18 575	200 324
Alloc Mem – total [B]	12 670 028	12 667 196	12 667 340	12 696 116
Alloc Mem – Trie [B]	12 666 356	<–	<–	<–
Alloc Mem – Buffers [B]	3 672	840	984	29 760
Energy – total [nJ]	13 977 999	13 746 268	13 858 686	16 273 222

Text	economist	economistShort	economistLong	bbc
Accesses – total	1 051 830	1 039 609	1 102 110	1 032 096
Acc – Scen1	1 024 608	<–	<–	1 024 608
Acc – Scen2	27 222	15 001	77 502	7 488
Alloc Mem – total [B]	12 671 396	12 667 772	12 674 372	12 667 364
Alloc Mem – Trie [B]	12 666 356	<–	<–	12 666 356
Alloc Mem – Buffers [B]	5 040	1 416	8 016	1 008
Energy – total [nJ]	13 973 561	13 811 205	14 641 531	13 711 395

Table 4.6: Initial Array-structured Trie Implementation – Small Dictionary

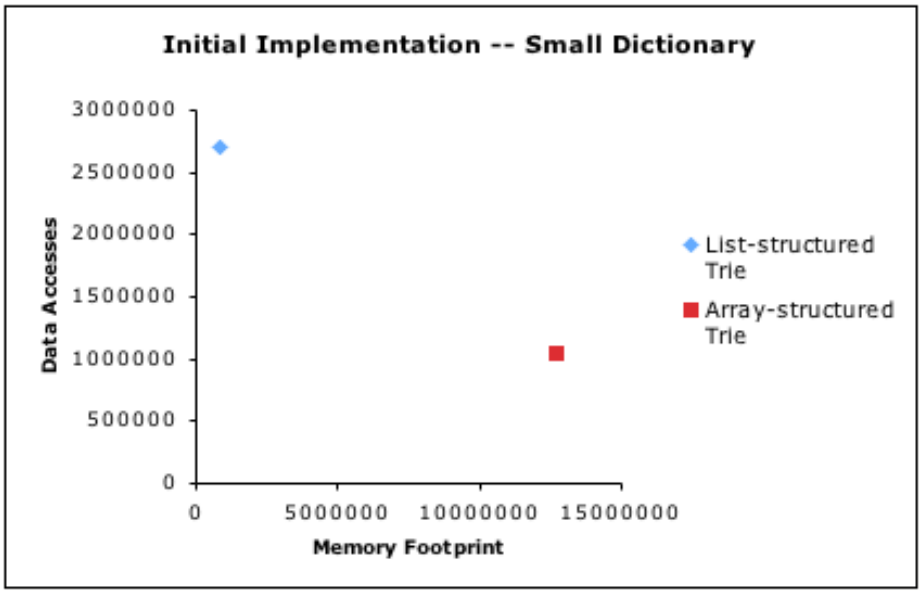

Figure 4.4: Initial Implementation – Array-, List-structured Implementation Compared

4.2.3 Transformation 1 – Split Container

The Split Container Transformation splits the dictionary into two parts according to a splitting rule. In this case, the splitting is driven by the occurrence frequency of a word in the English language. The dictionary is split into one small dictionary (Cont1) holding the most common words and a larger dictionary (Cont2) containing the rest of the original dictionary. The assignment of one word to one of the dictionaries is based on the frequency labels provided in the word list by [14].

In implementation terms, this means that two separate Tries are created. The first Trie (in the following referred to as Cont1 or Container1) consists of 3 219 popular words and the second Trie (referred to as Cont2 or Container2) holds the rest of the words (i.e., 53 827). For the small dictionary the first Trie consists again of 3 219 words and the second Trie consists of 17 614 words.

Table 4.7 and Table 4.8 contain the results for the large original dictionary. Table 4.9 and Table 4.10 present the results for the reduced dictionary.

In the following, the results for the list-structured Trie implementation are explained, followed by an explanation of the array-structured Trie implementation results.

Results for the List-structured Trie

Comparing the overall number of data accesses, after the Split Container transformation is applied, to the initial implementation, there is a slight global decrease in accesses

noticeable. There are various factors influencing this outcome.

First, there is an local increase in accesses due to the creation of two Tries. And while in a single Trie each word is only checked once, in the two Trie implementation a word is checked twice in the worst case.

Secondly however, large reductions in data accesses result from the fact that most of the accesses in Scenario2 (search) are to the Cont1. This is easily explained by the fact that texts indeed mainly consist of a large number of common words. So (for the spiegel example) only 294 out of 1107 words (i.e., 26,5% of the whole text) are not found in the small dictionary. This means that only 26,5% of the text have to be looked up in a very large dictionary instead of the whole text (i.e., 1107 words). Therefore, the majority of the lookup operations are performed on Cont1, which is a smaller Trie, in which less accesses are needed to find a word.

To summarize, the global reduction in accesses from distributing most of the weight to the small dictionary, outnumber the additional local costs of setting up two dictionaries, instead of one.

In terms of memory footprint however there is only an increase to note. This is owed to less sharing of characters, since the dictionary is split among two Tries. While in one Trie many characters are reused, in two Tries this structure is taken apart and characters that were shared before now need to be created in both Tries. For example, the two words 'nut' and 'nutmeg' are stored in six nodes in one Trie, since the word 'nut' is contained in both words. If this is split and 'nut' is stored in one Trie, while 'nutmeg' is stored in a second Trie, this sharing is lost. Now, 'nut' needs three nodes in the first Trie and 'nutmeg' allocates six additional nodes.

Although data accesses are only decreased slightly (2,5% for the *spiegel* text and the small dictionary) and memory footprint is increased 10,5% (Text: *spiegel*, small dictionary), energy consumption is largely decreased (42% for the *spiegel* text looked up in the small dictionary) compared to the initial list-structured Trie implementation. The results are depicted in Figure 4.5. For Scenario2 (search) the energy gains are between 67% and 81%.

This is only possible because the two Tries are assigned to two different memories. The first Trie (Cont1) resides on a small, energy efficient memory, while the second Trie (Cont2) is assigned to a larger, less energy efficient memory.

While Cont1 of the original dictionary can fit on a 256 kB SRAM, Cont2 of the original dictionary requires a 4MB SRAM. Overall energy consumption is calculated by adding the energy consumed by the small SRAM and the big SRAM. The energy of each SRAM is calculated by multiplying the total accesses to the container in each SRAM by the energy per access figure, which corresponds to the size of the SRAM [23].

Results for the Array-structured Trie

The above made observations are only partly valid for the array-structured Trie implementation.

Here, it is to note that the number of accesses increases for Scenario1, as well as Scenario2, which results into an overall increase in accesses. For example, looking up the text *spiegel* in the small dictionary, implemented with array-structured Tries,

increases the number of overall accesses by 0,9% compared to the accesses it takes to lookup the same text in the initial implementation. This is explained by the fact that an array-structured Trie implementation is already very efficient in accesses, due to the direct access in the arrays. Therefore, only the increase in accesses (as explained above on the list-structured Trie implementation) takes effect.

The overall memory footprint is increased due to less character reuse in the dictionaries. Here the overall memory footprint of the application increases by 10,5% for the small dictionary, comparing it to the initial implementation.

Energy results can not be provided for the implementation using the original dictionary. Here Cont2 exceeds the possible SRAM size of 16 MB.

For a reduced dictionary, it is however interesting to note, that there is a large gain in energy compared to the initial array-structured Trie implementation. Looking up the *spiegel* text is 44,2% more energy efficient than it is in the initial implementation.

This again is due to the fact that the two Tries are assigned to two differently sized memories, where the smaller memory is more energy efficient. For the reduced dictionary Cont1 requires a 4MB SRAM while Cont2 is assigned to a 16MB SRAM. The overall energy consumption is calculated as described above for the list-structured Trie.

Text	spiegel	independent	theindependent	spiegelLong
SplitCont	(294: 132)	(90:26)	(155:34)	(3269:1072)
Accesses – total	8 001 065	7 921 568	7 954 908	8 777 946
Scenario1 – total	7 878 609	<–	<–	<–
Cont1 – Scen1	205 077	<–	<–	<–
Cont2 – Scen1	4 767 908	<–	<–	<–
Buffers-Scen1	2 905 624	<–	<–	<–
Scenario2 – total	122 456	42 959	76 299	899 337
Cont1 – Scen2	54 058	23 517	36 266	376 637
Cont2 – Scen2	20 963	2 938	10 710	174 224
Buffers-Scen2	47 435	16 504	29 323	348 476
Alloc. Memory – total[B]	2 481 784	2 479 240	2 479 432	2 504 344
Container1 [B]	139 056	<–	<–	<–
Container2 [B]	2 339 536	<–	<–	<–
Buffers [B]	3 192	648	840	25 752
Energy – total [nJ]	23 005 640	22 929 396	22 692 212	24 709 075
En. Gain-total	40.1%	39.75%	40.6%	40.78%
Energy – Scen1 [nJ]	22 951 555	22 890 537	22890537	23445512
Energy – Scen2 [nJ]	153 554	36 937	86 374	1 263 563
Energy Gain – Scenario2	72%	81%	76%	67%

Text	economist	economistShort	economistLong	bbc
SplitCont	(344:172)	(154:41)	(786:256)	(90:36)
Accesses – total	7 998 926	7 943 079	8 206 843	7 912 756
Scenario1 – total	7 878 609	<–	<–	<–
Cont1 – Scen1	205 077	<–	<–	<–
Cont2 – Scen1	4 767 908	<–	<–	<–
Buffers-Scen1	2 905 624	<–	<–	<–
Scenario2 – total	120 317	64 470	328 234	34 147
Cont1 – Scen2	50 744	29 263	145 414	18 212
Cont2 – Scen2	22 561	11 008	55 676	2 789
Buffers-Scen2	47 012	24 199	127 144	13 146
Alloc. Memory – total[B]	2 648 696	<–	<–	<–
Container1 [B]	139 056	<–	<–	<–
Container2 [B]	2 339 536	<–	<–	<–
Buffers [B]	4 152	1 008	6 168	888
Energy – total [nJ]	23 177 399	22 971 473	23 430 956	22 921 588
En. Gain-total	39.61%	39.78%	40.44%	39.69%
Energy – Scen1 [nJ]	23 018 385	22 890 537	23 018 385	22 890 537
Energy – Scen2 [nJ]	159 014	80 935	412 571	46 365
Energy Gain – Scenario2	69%	54 72%	72%	69%

Table 4.7: List-structured Trie Implementation – Split Container Transformation – Original Dictionary

Text	spiegel	independent	theindependent	spiegelLong
Accesses – total	3 117 663	3 096 906	3 106 563	3 324 163
Scen1 total	3 085 336	3 085 336	<–	<–
Scen1. Cont1	395 545	395 545	<–	<–
Scen1. Cont2	1 222 414	1 222 414	<–	<–
Scen1. Buffers	1 467 377	<–	<–	<–
Scen2. total	32 327	11 570	21 227	238 827
Scen2. Cont1	8 921	3 444	6 512	62 415
Scen2. Cont2	3 424	1 161	1 963	28 840
Scen2. Buffers	19 982	6 965	12 752	147 572
Alloc. Mem – total	36 163 820	36 161 276	36 161 468	36 186 380
Cont1	2 051 076	<–	<–	<–
Cont2	34 109 552	<–	<–	<–
Buffers	3 192	648	840	25 752
Energy – total [nJ]	—	—	—	—

Text	economist	economistShort	economistLong	bbc
Accesses – total	3 118 214	3 102 909	3 176 650	3 094 207
Scen1 total	3 085 336	<–	<–	<–
Scen1. Cont1	395 545	<–	<–	<–
Scen1. Cont2	1 222 414	<–	<–	<–
Scen1. Buffers	1 467 377	<–	<–	<–
Scen2. total	32 878	17 573	91 314	8 871
Scen2. Cont1	8 180	4 842	25 101	2 422
Scen2. Cont2	4 189	2 067	10 615	1 000
Scen2. Buffers	20 509	10 664	55 598	5 449
Alloc. Mem – total	36 164 780	36 161 636	36 166 796	36 161 516
Cont1	2 051 076	<–	<–	<–
Cont2	34 109 552	<–	<–	<–
Buffers	4 152	1 008	6 168	888
Energy – total [nJ]	—	—	—	—

Table 4.8: Array-structured Trie Implementation – Split Container Transformation – Original Dictionary

Text	spiegel	independent	theIndependent	spiegelLong
Accesses – total	2 646 210	2 568 879	2 601 732	3 400 178
Acc – Scen1 total	2 526 942	<–	<–	<–
Scen1 – Cont1	205 077	<–	<–	<–
Scen1 – Cont2	1 397 901	<–	<–	<–
Scen2 – Buffers	923 964	<–	<–	<–
Acc – Scen2 total	119 268	41 937	74 790	873 236
Scen2 – Cont1	54 058	20 050	39 591	376 637
Scen2 – Cont2	18 649	5 634	6 332	154 904
Scen2 – Buffers	46 561	16 253	28 867	341 695
Alloc Mem – total [B]	953 144	950 312	950 456	979 232
Alloc Mem – Cont1 [B]	139 056	<–	<–	<–
Alloc Mem – Cont2 [B]	810 416	<–	<–	<–
Alloc Mem – Buffers [B]	3 672	840	984	29 760
Energy – total [nJ]	2 975 648	2 929 889	2 917 611	3 527 252
Energy Gain [%]	42	41,5	42,4	44,6

Text	economist	economistShort	economistLong	bbc
Accesses – total	2 643 058	2 589 835	2 846 597	2 560 338
Acc – Scen1 total	2 526 942	<–	<–	<–
Scen1 – Cont1	205 077	<–	<–	<–
Scen1 – Cont2	1 397 901	<–	<–	<–
Scen1 – Buffers	923 964	<–	<–	<–
Acc – Scen2 total	116 116	62 893	319 655	33 396
Scen2 – Cont1	50 744	32 005	145 414	14 964
Scen2 – Cont2	19 416	6 395	49 131	5 484
Scen2 – Buffers	45 956	24 493	125 110	12 948
Alloc Mem – total [B]	954 512	950 888	957 488	950 480
Alloc Mem – Cont1 [B]	139 056	<–	<–	<–
Alloc Mem – Cont2 [B]	810 416	<–	<–	<–
Alloc Mem – Buffers [B]	5 040	1 416	8 016	1 008
Energy – total [nJ]	2 996 197	2 917 923	3 148 837	2 891 956
Energy Gain [%]	41,5	42,1	42,6	42

Table 4.9: List-structured Trie Implementation – Transformation Split Container – Small Dictionary

Text	spiegel	independent	theIndependent	spiegelLong
Accesses – total	1 062 633	1 041 903	1 051 487	1 268 965
Acc – Scen1 total	1 030 278	<–	<–	<–
Scen1 – Cont1	397 046	<–	<–	<–
Scen1 – Cont2	471 409	<–	<–	<–
Scen2 – Buffers	161 823	<–	<–	<–
Acc – Scen2 total	32 355	11 625	21 209	238 687
Scen2 – Cont1	8 921	3 444	6 512	62 415
Scen2 – Cont2	3 256	1 110	1 911	26 988
Scen2 – Buffers	20 178	7 071	12 786	149 284
Alloc Mem – total [B]	14 008 384	14 005 552	14 005 696	14 034 472
Alloc Mem – Cont1 [B]	2 051 076	<–	<–	<–
Alloc Mem – Cont2 [B]	11 953 636	<–	<–	<–
Alloc Mem – Buffers [B]	3 672	840	984	29 760
Energy – total [nJ]	7 794 270	7 741 655	7 763 619	8 363 287
Energy Gain [%]	44,2	43,7	44	48,6

Text	economist	economistShort	economistLong	bbc
Accesses – total	1 062 900	1 047 933	1 121 691	1 039 158
Acc – Scen1 total	1 030 278	<–	<–	<–
Scen1 – Cont1	397 046	<–	<–	<–
Scen1 – Cont2	471 409	<–	<–	<–
Scen1 – Buffers	161 823	<–	<–	<–
Acc – Scen2 total	32 622	17 655	91 413	8 880
Scen2 – Cont1	8 180	4 842	25 101	2 422
Scen2 – Cont2	3 727	1 939	9 868	959
Scen2 – Buffers	20 715	10 874	56 444	5 499
Alloc Mem – total [B]	14 009 752	14 006 128	14 012 728	14 005 720
Alloc Mem – Cont1 [B]	2 051 076	<–	<–	<–
Alloc Mem – Cont2 [B]	11 953 636	<–	<–	<–
Alloc Mem – Buffers [B]	5 040	1 416	8 016	1 008
Energy – total [nJ]	7 828 881	7 774 781	8 000 794	7 743 247
Energy Gain [%]	44	43,7	45,3	43,5

Table 4.10: Array-structured Trie Implementation – Transformation Split Container – Small Dictionary

4.2.4 Transformation 2 – Remove Redundant Attribute

The Remove Redundant Attribute Transformation suggests the removal of a substitutable attribute. The *end* attribute is such an attribute. It is stored in each node of the

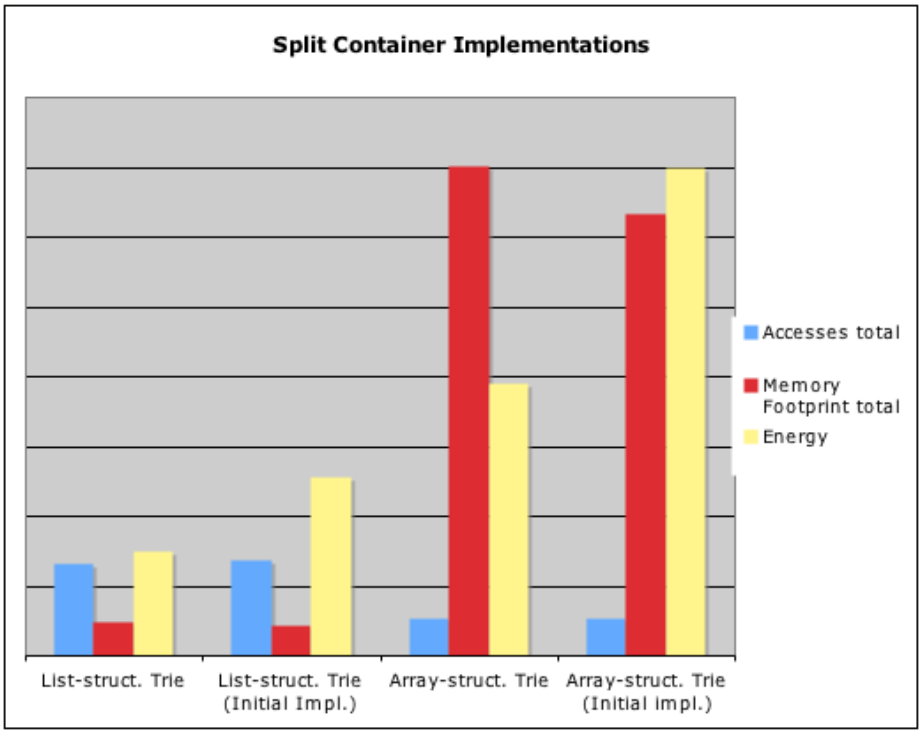

Figure 4.5: Split Container – Array-, List-structured Trie Compared, Text: spiegel, Small Dictionary

Trie to check whether the end of a word is reached or not. Removing *end* accounts to removing a 4 Byte integer from each node in the list-structured implementation. In the array-structured implementation it results in removing a 26 cell integer array (104 Byte). The concrete algorithmic changes, that this transformation imposes, are explained in detail in Section 2.8.2.

Table 4.11 and Table 4.12 hold the results for the original large dictionary, while Table 4.13 and Table 4.14 show the results for the reduced dictionary.

For both implementations the number of accesses is reduced. Less accesses are needed to setup the dictionary, since the *end* attribute does not need to be written. The overall number of accesses is reduced by 15,4% for looking up the *spiegel* text in the reduced dictionary implemented with an array-structured Trie. Figure 4.7 depicts the results for data accesses, memory footprint and energy consumption for the *spiegel* text looked up in the small dictionary implemented with a array-structured Trie. (Figure 4.6 shows the corresponding results for the list-structured Trie implementation.)

Also the memory footprint is reduced. As mentioned above, for the list-structured Trie 4 Bytes and for the array-structured Trie 104 Bytes are saved per node. For looking up the *spiegel* text in the small dictionary, implemented with an array-structured Trie,

this accounts to a reduction of 44%.

The energy gains for the list-structured implementation vary from 24,6% for the original large dictionary to an average of 2% for the reduced dictionary. For the array-structured Trie no energy calculations are made for the original dictionary implementation, since here again the 16 MB SRAM size is exceeded. For a reduced dictionary, however, the gains amount to an average of 46,6%.

Text	spiegel	independent	theindependent	spiegelLong
Accesses-total	7 946 789	7 872 837	7 906 260	8 645 107
Scenario1	7 830 836	<–	<–	<–
Scenario2	115 353	42 001	75 424	814 271
Alloc. Memory - total	1 785 288	1 782 744	1 782 936	1 807 848
Trie	1 782 096	<–	<–	<–
Buffers	3 192	648	840	25 752
Energy-total [nJ]	28 973 993	28 704 364	28 826 224	31 520 060
Gain	24.6%	24.6%	24.6%	24.6%
Scen2-Energy [nJ]	420 577	153 136	274 996	2 968 832
Scen2-Gain	23.2%	23.2%	23.2%	23.2%

Text	economist	economistShort	economistLong	bbc
Accesses-total	7 939 966	7 891 259	8 141 808	7 862 328
Scenario1	7 830 836	<–	<–	<–
Scenario2	109 130	60 423	310 972	31 492
Alloc. Memory - total	1 786 248	1 783 104	1 788 264	1 782 984
Trie	1 782 096	<–	<–	<–
Buffers	4 152	1 008	6 168	888
Energy-total [nJ]	28 949 116	28 771 530	29 685 032	28 666 048
Gain	24.6%	24.6%	24.6%	24.6%
Scen2-Energy [nJ]	397 888	220 302	1 133 804	114 820
Scen2-Gain	23.2%	23.2%	23.2%	23.2%

Table 4.11: List-structured Trie Implementation – Remove Redundant Attribute Transformation – Original Dictionary

Text	spiegel	independent	theindependent	spiegelLong
Accesses – total	2 618 180	2 600 718	2 609 249	2 790 992
Scenario1	2 590 664	2 590 664	<–	<–
Scenario2	27 516	10 054	18 585	200 328
Alloc. Mem – total [B]	19 380 264	19 377 720	19 377 912	19 402 824
Trie [B]	19 377 072	<–	<–	<–
Buffers [B]	3 192	648	840	25 752
Energy - total [nJ]	—	—	—	—

Text	economist	economistShort	economistLong	bbc
Accesses - total	2 618 134	2 605 583	2 667 991	2 598 139
Scenario1	2 590 664	<–	<–	<–
Scenario2	27 470	14 919	77 327	7 475
Alloc. Mem – total [B]	19 381 224	19 378 080	19 383 240	19 377 960
Trie [B]	19 377 072	<–	<–	<–
Buffers [B]	4 152	1 008	6 168	888
Energy – total [nJ]	—	—	—	—

Table 4.12: Array-structured Trie Implementation – Remove Redundant Attribute Transformation – Original Dictionary

Text	spiegel	independent	theIndependent	spiegelLong
Accesses – total	2 658 538	2 589 978	2 621 577	3 310 618
Acc – Scen1	2 550 762	<–	<–	<–
Acc – Scen2	107 774	39 214	70 813	759 854
Alloc Mem – total [B]	647 724	644 862	645 036	673 812
Alloc Mem – Trie [B]	644 052	<–	<–	<–
Alloc Mem – Buffers [B]	3 672	840	984	29 760
Energy – total [nJ]	5 035 271	4 905 418	4 965 267	6 270 310
Energy Gain [%]	2	2.1	2.1	1,6

Text	economist	economistShort	economistLong	bbc
Accesses – total	2 651 975	2 607 194	2 840 957	2 580 257
Acc – Scen1	2 550 762	<–	<–	2 550 762
Acc – Scen2	101 211	56 430	290 193	29 493
Alloc Mem – total [B]	649 092	645 468	652 068	645 060
Alloc Mem – Trie [B]	644 052	<–	<–	<–
Alloc Mem – Buffers [B]	5 040	1 416	8 016	1 008
Energy – total [nJ]	5 022 841	4 938 025	5 380 772	4 887 007
Energy Gain [%]	2	2.1	1.9	2.1

Table 4.13: List-structured Trie Implementation – Transformation Remove Redundant – Small Dictionary

Text	spiegel	independent	theIndependent	spiegelLong
Accesses – total	890 341	872 898	881 360	1 063 109
Acc – Scen1	862 785	<–	<–	<–
Acc – Scen2	27 556	10 113	18 575	200 324
Alloc Mem – total [B]	7 088 244	7 085 412	7 085 556	7 114 332
Alloc Mem – Trie [B]	7 084 572	<–	<–	<–
Alloc Mem – Buffers [B]	3 672	840	984	29 760
Energy – total [nJ]	7 447 702	7 301 792	7 372 576	8 892 907
Energy Gain [%]	46.7	46.9	46.9	45.4

Text	economist	economistShort	economistLong	bbc
Accesses – total	890 007	877 786	940 287	870 273
Acc – Scen1	862 785	<–	<–	862 785
Acc – Scen2	27 222	15 001	77 502	7 488
Alloc Mem – total [B]	7 089 612	7 085 988	7 092 588	7 085 580
Alloc Mem – Trie [B]	7 084 572	<–	<–	<–
Alloc Mem – Buffers [B]	5 040	1 416	8 016	1 008
Energy – total [nJ]	7 444 908	7 342 680	7 865 501	7 279 834
Energy Gain [%]	46.8	46.9	46.3	47

Table 4.14: Array-structured Trie Implementation – Transformation Remove Redundant – Small Dictionary

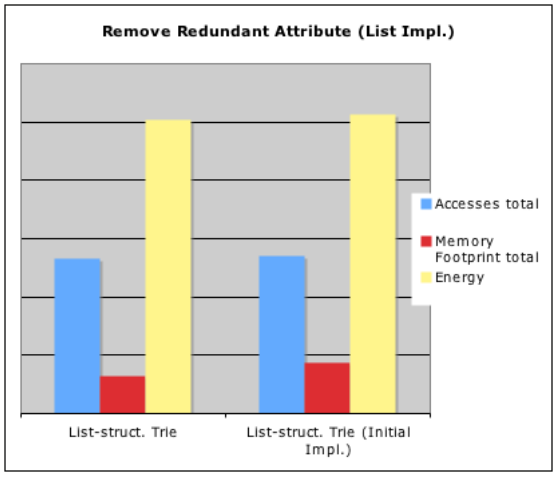

Figure 4.6: Remove Redundant Attribute – List-structured Trie Compared, Text: spiegel, Small Dictionary

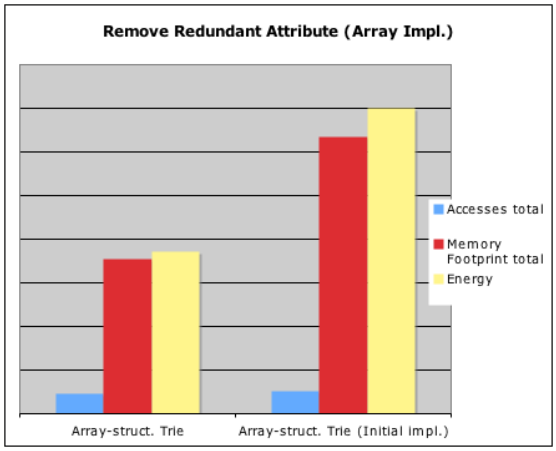

Figure 4.7: Remove Redundant Attribute – Array-structured Trie Compared, Text: spiegel, Small Dictionary

4.2.5 Transformation 3 – Change Relationship Ordering

The transformation Change Relationship Ordering changes the ordering of the relationship between *Dictionary* and *Word* from alphabetically ordered [a..z] to an ordering according to letter probability[2]. This means that in the list-structured Trie the characters in horizontal alignment are not ordered alphabetically as shown in Figure 4.2. Instead the overall letter probability in the English language is used. For example, the first four letters now are {e, t, a, o} instead of {a, b, c, d} before.

This transformation is only implemented for the list-structured Trie , since no gains are expected for an array implementation. The reason is that the array implementation has direct access to its components in both cases and is thus not expected to profit from faster sequential search. Table 4.15 and Table 4.16 present the results of this transformation. Figure 4.8 shows accesses, memory footprint and energy results for the Change Ordering transformation compared to the initial implementation.

The total number of data accesses is reduced by 29,2% comparing the small dictionary to the initial small dictionary implementation. For the setup of the dictionary (Scenario1) less accesses are used, since in average the position to insert a word is found faster. This amounts to a decrease of 29,8% of data accesses in Scenario1. Also the number of accesses for a search (Scenario2) is reduced because, here again, the requested word is found faster and less nodes need to traversed. This results in a decrease of 15,6% in memory accesses.

The memory footprint does not change, since the number of elements stored is not changed in this transformation.

In terms of energy the average gain for the original dictionary is 30%.

[2] {e, t, a, o, i, n, s, h, r, d, l, c, u, m, w, f, g, y, p, b, v, k, j, x, q, z}

Text	spiegel	independent	theindependent	spiegelLong
Accesses – total	5 631 141	5 570 361	5 599 783	6 211 841
Scen1	5 535 052	<–	<–	<–
Scen2	96 089	35 309	64 731	676 789
Alloc. memory total [B]	2 379 320	2 376 776	2 376 968	2 401 880
Trie [B]	2 376 128	<–	<–	<–
Buffers [B]	3 192	648	840	25 752
Energy total [nJ]	26 719 764	26 431 363	26 570 970	29 475 185
Energy Gain [%]	30.4	30.6	30.5	29.4
Scen1 [nJ]	26 263 822	<–	<–	<–
Gain Scen1 [%]	30.6	<–	<–	<–

Text	economist	economistShort	economistLong	bbc
Accesses – total	5 628 371	5 586 484	5 798 954	5 561 234
Scen1	5 535 052	<–	<–	<–
Scen2	93 319	51 432	263 902	26 182
Alloc. memory total [B]	2 380 280	2 377 136	2 382 296	2 377 016
Trie [B]	2 376 128	<–	<–	<–
Buffers [B]	4 152	1 008	6 168	888
Energy total [nJ]	26 706 620	26 507 866	27 516 037	26 388 055
Energy Gain [%]	30.4	30.5	30.1	30.6
Scen1 [nJ]	26 263 822	<–	<–	<–
Gain Scen1 [%]	30.6	<–	<–	<–

Table 4.15: List-structured Trie Implementation – Change Ordering Transformation – Original Dictionary

Text	spiegel	independent	theIndependent	spiegelLong
Accesses – total	1 918 830	1 861 321	1 889 568	2 470 712
Acc – Scen1 total	1 827 838	<–	<–	<–
Acc – Scen2 total	90 989	33 480	61 727	642 871
Alloc Mem – total [B]	862 408	859 576	859 720	888 496
Alloc Mem – Trie [B]	858 736	<–	<–	<–
Alloc Mem – Buffers [B]	3 672	840	984	29 760
Energy – total [nJ]	3 634 264	3 525 342	3 578 842	4 679 528
Energy Gain [%]	29.3	29.6	29.4	26.6

Text	economist	economistShort	economistLong	bbc
Accesses – total	1 915 516	1 876 777	2 078 199	1 852 799
Acc – Scen1 total	1 827 838	<–	<–	<–
Acc – Scen2 total	87 675	48 936	250 358	24 958
Alloc Mem – total [B]	863 776	860 152	866 752	859 744
Alloc Mem – Trie [B]	858 736	<–	<–	<–
Alloc Mem – Buffers [B]	5 040	1 416	8 016	1 008
Energy – total [nJ]	3 627 987	3 554 616	3 936 109	3 509 201
Energy Gain [%]	29.3	29.5	28.3	29.7

Table 4.16: List-structured Trie Implementation – Transformation Change Ordering – Small Dictionary

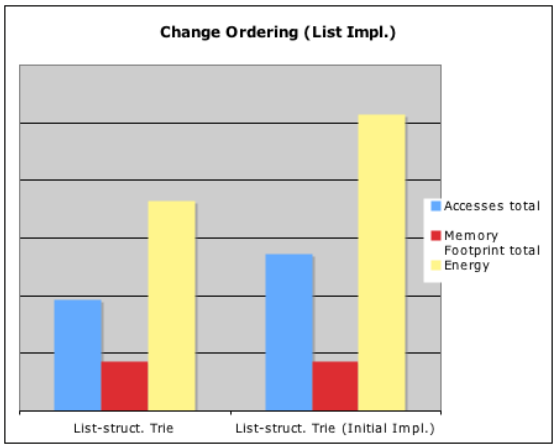

Figure 4.8: Change Relationship Ordering – List-structured Trie Compared, Text: spiegel, Small Dictionary

4.2.6 Transformation 4 – Attribute to Class

According to the Attribute to Class transformation, an attribute is transformed into a class. For several reasons this is beneficial. On the one hand, this class can be subject to new transformations that can not be applied to its former attributes. The gains of this are not shown here, however. On the other hand, the newly created object can be suggested to reside on a different memory. The gains for this are presented here. The third reason is based on the input texts. As part of Section 4.1.2 'Input Characterization' it is shown that errors have re-occurrence rates up to 50%. This means a word, which is misspelled once is likely to be misspelled again. In the context of this, it is beneficial to consult first the list of misspelled words before the whole dictionary is searched. The results for this transformation are also presented here.

In this implementation the *misspelledList* attribute in *DictionaryEngine* is transformed into the class *MisspelledList*. The list of misspelled words that was before the value of *misspelledList*, is now the attribute of the class *MisspelledList*. Now *MisspelledList* is not only filled with erroneous words, it is also consulted before the dictionary is consulted. In implementation terms this means that a new type *MisspelledList* is created.

Table 4.17 and Table 4.19 present the results for the list-structured Trie implementation, while Table 4.18 and Table 4.20 show the results for the array-structured Trie.

The total number of data accesses is increased. This is due to the increase of accesses in Scenario2, here checking every word first in the misspelled list creates overhead. This suggests that looking up each word first in the misspelled list, does not create any gains. For larger input texts and ongoing storage of misspelled words there are still gains in data accesses expected however. The number of data accesses for Scenario1 remain constant, since words are added to the *MisspelledList*, only after search and nothing is changed in the setup.

The overall memory footprint remains also constant. This is true, because the memory footprint considers the filled misspelled list at the end of the application's execution.

For energy calculations two version are considered. Version one calculates the energy for all data types assigned to one memory. The second version (depicted with 'Trie vs. Buffers') assigns the newly created type to one memory and the full Trie to a second. The shown gains arise comparing the results of the latter energy calculation ('Trie vs. Buffers') to the results of the original implementation in Table 4.3.

For the list-structured Trie implementation, the average energy gain is 35,7% for the original dictionary. Figure 4.9 shows the results for accesses, memory footprint and energy consumption ('Trie vs. Buffers') of looking up the *spiegel* text in the small dictionary implemented with a list-structured Trie.

Text	spiegel	independent	theindependent	spiegelLong
Accesses – total	8 192 125	8 033 982	8 082 217	12 756 288
Scen1 – total	7 979 343	<–	<–	<–
Scen1 MisspList	0	<–	<–	<–
Scen1 Trie	5 033 733	<–	<–	<–
Scen1 Buffer	2 945 610	<–	<–	<–
Scen2 – total	212 779	54 636	102 871	4 776 942
Scen2 MisspList	52 350	6 562	14 142	2 023 756
Scen2 Trie	67 638	25 932	46 424	457 078
Scen2 Buffer	92 791	22 142	42 305	2 296 108
Alloc Mem [B] – total [B]	2 379 320	2 376 776	2 376 968	2 401 880
Trie [B]	2 376 128	<–	<–	<–
Buffers [B]	3192	648	840	25 752
Energy – total [nJ]	38 871 633	38 121 244	38 350 120	60 528 586
– Trie vs. Buffers [nJ]	24 345 089	24 079 494	24 177 394	27 615 975
Gain Trie vs. Buffers[%]	36,6	36,7	36,7	33,8

Text	economist	economistShort	economistLong	bbc
Accesses - total	8 203 611	8 062 681	8 962 886	8 020 509
Scen1 - total	7 979 343	<–	<–	<–
Scen1 MisspList	0	<–	<–	<–
Scen1 Trie	5 033 733	<–	<–	<–
Scen1 Buffer	2 945 610	<–	<–	<–
Scen2 - total	224 265	83 335	983 540	41 163
Scen2 MisspList	62 459	12 058	341 407	5 593
Scen2 Trie	95 052	36 973	525 643	18 735
Scen2 Buffer	66 754	34 304	116 490	16 835
Alloc Mem [B] – total [B]	2 380 280	2 377 136	2 382 296	<–
Trie [B]	2 376 128	<–	<–	<–
Buffers [B]	4 152	1 008	6 168	888
Energy - total [nJ]	38 926 134	38 257 421	42 528 894	38 057 315
– Trie vs. Buffers [nJ]	24 545 173	24 132 307	26 610 678	24 045 194
Gain [%]	36	36,7	32,2	36,7

Table 4.17: List-structured Trie Implementation – Attribute to Class Transformation – Original Dictionary

Text	spiegel	independent	theindependent	spiegelLong
Accesses – total	3 198 377	3 099 628	3 123 734	6 940 614
Scen1 – total	3 078 910	<–	<–	<–
Scen1 MisspList	0	<–	<–	<–
Scen1 Trie	1 611 533	<–	<–	<–
Scen1 Buffer	1 467 377	<–	<–	<–
Scen2 – total	119 467	20 718	44 824	3 861 704
Scen2 MisspList	47 650	5 526	13 447	1 845 983
Scen2 Trie	9 669	3 891	7 193	72 493
Scen2 Buffer	62 148	1 1301	24 184	1 943 228
Alloc Mem – total [B]	34 647 048	34 644 504	34 644 696	34 669 608
Trie [B]	34 643 856	<–	<–	<–
Buffers [B]	3 192	648	840	25 752

Text	economist	economistShort	economistLong	bbc
Accesses – total	3 216 394	3 115 611	3 787 108	3 096 269
Scen1 – total	3 078 910	<–	<–	<–
Scen1 MisspList	0	<–	<–	<–
Scen1 Trie	1 611 533	<–	<–	<–
Scen1 Buffer	1 467 377	<–	<–	<–
Scen2 – total	137 484	36 701	708 198	17 359
Scen2 MisspList	57 362	11 375	318 648	5 313
Scen2 Trie	9 156	5 633	28 621	2 682
Scen2 Buffer	70 966	19 693	360 929	9 364
Alloc Mem – total [B]	34 648 008	34 644 864	34 650 024	34 644 744
Trie [B]	34 643 856	<–	<–	<–
Buffers [B]	4 152	1 008	6 168	888

Table 4.18: Array-structured Trie Implementation – Array MisspelledList as Class – Original Dictionary

Text	spiegel	independent	theIndependent	spiegelLong
Accesses – total	2 833 804	2 659 370	2 706 839	8 471 827
Acc – Scen1 total	2 604 432	<–	<–	<–
Acc – Scen2 total	229 369	54 935	102 404	5 867 392
Alloc Mem – total [B]	862 408	859 576	859 720	888 496
Alloc Mem – Trie [B]	858 736	<–	<–	<–
Alloc Mem – Buffers [B]	3 672	840	984	29 760
Energy – total [nJ]	5 367 225	5 036 847	5 126 753	16 045 640
– Trie vs. Buffers [nJ]	4 351 034	4 116 424	4 180 176	10 072 739
Energy Gain [%]	15.3	17.8	17.6	-58

Text	economist	economistShort	economistLong	bbc
Accesses – total	2 855 140	2 694 088	3 799 408	2 645 484
Acc – Scen1 total	2 604 432	<–	<–	<–
Acc – Scen2 total	250 705	89 653	1 194 973	41 049
Alloc Mem – total [B]	863 776	860 152	866 752	859 744
Alloc Mem – Trie [B]	858 736	<–	<–	<–
Alloc Mem – Buffers [B]	170 100	<–	<–	<–
Energy – total [nJ]	5 407 635	5 102 603	7 196 079	5 010 547
– Trie vs. Buffers [nJ]	4 336 146	4 159 324	5 350 630	4 097 056
Energy Gain [%]	15.4	17.5	2.5	17.9

Table 4.19: List-structured Trie Implementation – Transformation Attribute To Class – Small Dictionary

Text	spiegel	independent	theIndependent	spiegelLong
Accesses – total	1 169 193	1 049 401	1 073 539	6 138 518
Acc – Scen1 total	1 024 608	<–	<–	<–
Acc – Scen2 total	144 585	24 793	48 931	5 113 910
Alloc Mem – total [B]	12 670 028	12 667 196	12 667 340	12 696 116
Alloc Mem – Trie [B]	12 666 356	<–	<–	<–
Alloc Mem – Buffers [B]	3 672	840	984	29 760
Energy – total [nJ]	15 532 729	13 941 292	14 261 966	81 550 212
– Trie vs. Buffers [nJ]	11 873 721	11 688 527	11 752 688	17 363 295
Energy Gain [%]	15.1	15	15.2	- 6.3

Text	economist	economistShort	economistLong	bbc
Accesses – total	1 196 134	1 072 379	1 979 028	1 043 761
Acc – Scen1 total	1 024 608	<–	<–	<–
Acc – Scen2 total	171 526	47 771	954 420	19 153
Alloc Mem – total [B]	12 671 396	12 667 772	12 674 372	12 667 364
Alloc Mem – Trie [B]	12 666 356	<–	<–	<–
Alloc Mem – Buffers [B]	5 040	1 416	8 016	1 008
Energy – total [nJ]	15 890 640	14 246 555	26 291 387	13 866 365
– Trie vs. Buffers [nJ]	11 891 182	11 731 276	12 877 243	11 668 663
Energy Gain [%]	15	15.1	12.1	14.9

Table 4.20: Array-structured Trie Implementation – Transformation Attribute To Class – Small Dictionary

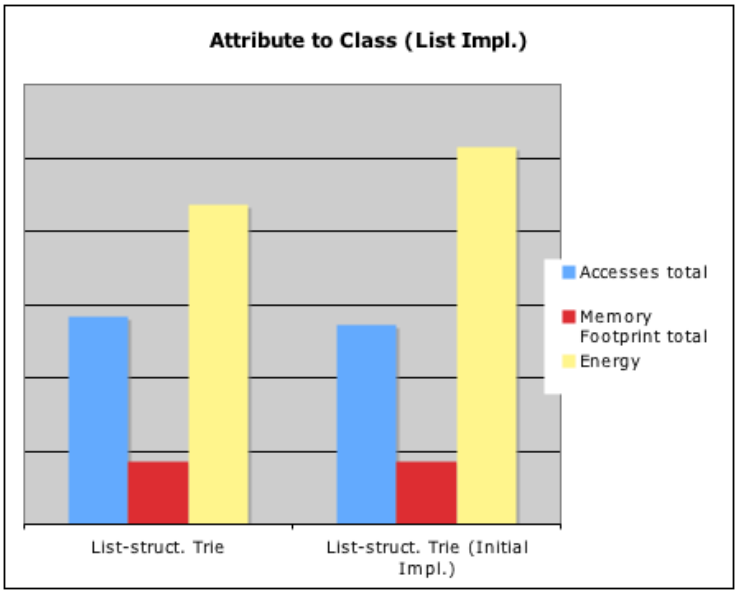

Figure 4.9: Attribute to Class – List-structured Trie Compared, Text: spiegel, Small Dictionary

4.2.7 Transformations – Split Container, Remove Redundant, Change Ordering – combined

In this implementation the transformations Split Container, Remove Redundant Attribute and Change Ordering are combined. Table 4.21 and Table 4.22 present the results for this combined transformation for the reduced dictionary.

The overall number of data accesses is reduced, as well as the memory footprint. For checking the *spiegel* text in the small dictionary, implemented with a list-structured Trie, the data accesses are reduced by 32,2% compared to the corresponding initial implementation. The memory footprint is decreased by 17% compared to the initial implementation.

Energy gains account to an average 63,1% for the list-structured implementation and an average 70% for the array-structured Trie implementation. Explanations for this behavior can be found in the aforementioned sections. Figure 4.10 shows the results for comparing accesses, memory footprint and energy consumption between the combined transformations (list-structured Trie, text: *spiegel*, small dictionary) and the initial implementation.

Text	spiegel	independent	theIndependent	spiegelLong
Accesses – total	1 838 823	1 771 151	1 801 859	2 499 200
Acc – Scen1 total	1 733 497	<–	<–	<–
Scen1 – Cont1	135 437	<–	<–	<–
Scen1 – Cont2	913 155	<–	<–	<–
Scen1 – Buffers	684 905	<–	<–	<–
Acc – Scen2 total	105 326	37 654	68 362	765 703
Scen2 – Cont1	79 795	20 058	31 278	313 805
Scen2 – Cont2	17 869	2 790	10 385	146 312
Scen2 – Buffers	7 662	14 806	26 699	305 586
Alloc Mem – total [B]	715 776	712 944	713 088	<–
Alloc Mem – Cont1 [B]	104 292	<–	<–	<–
Alloc Mem – Cont2 [B]	607 812	<–	<–	<–
Alloc Mem – Buffers [B]	3 672	840	984	29 760
Energy – total [nJ]	1 896 330	1 825 142	1 845 119	2 432 077
Energy Gain [%]	63	63,5	63,6	61,8

Text	economist	economistShort	economistLong	bbc
Accesses – total	1 839 730	1 789 817	2 021 160	1 763 030
Acc – Scen1 total	1 733 497	<–	<–	<–
Scen1 – Cont1	135 437	<–	<–	<–
Scen1 – Cont2	913 155	<–	<–	<–
Scen1 – Buffers	684 905	<–	<–	<–
Acc – Scen2 total	106 233	56 320	287 663	29 533
Scen2 – Cont1	77 732	27 745	123 124	12 516
Scen2 – Cont2	20 090	6 309	50 221	5 376
Scen2 – Buffers	8 411	28 575	114 318	11 641
Alloc Mem – total [B]	717 144	713 520	720 120	713 112
Alloc Mem – Cont1 [B]	104 292	<–	<–	<–
Alloc Mem – Cont2 [B]	607 812	<–	<–	<–
Alloc Mem – Buffers [B]	5 040	1 416	8 016	1 008
Energy – total [nJ]	1 915 540	1 841 481	2 001 281	1 826 397
Energy Gain [%]	62,6	63,5	63,5	63,4

Table 4.21: List-structured Trie Implementation – Transformation Combination – Small Dictionary

Text	spiegel	independent	theIndependent	spiegelLong
Accesses – total	900 810	880 080	889 664	1 107 142
Acc – Scen1 total	868 455	<–	<–	<–
Scen1 – Cont1	375 476	<–	<–	<–
Scen1 – Cont2	331 156	<–	<–	<–
Scen1 – Buffers	161 823	<–	<–	<–
Acc – Scen2 total	32 355	11 625	21 209	238 687
Scen2 – Cont1	8 921	3 444	6 512	62 415
Scen2 – Cont2	3 256	1 110	1 911	26 988
Scen2 – Buffers	20 178	7 071	12 786	149 284
Alloc Mem – total [B]	7 836 816	7 833 984	7 834 128	7 862 904
Alloc Mem – Cont1 [B]	1 147 212	<–	<–	<–
Alloc Mem – Cont2 [B]	6 685 932	<–	<–	<–
Alloc Mem – Buffers [B]	3 672	840	984	29 760
Energy – total [nJ]	4 207 058	4 165 001	4 183 041	4 659 313
Energy Gain [%]	70	69,7	69,8	71,4

Text	economist	economistShort	economistLong	bbc
Accesses – total	901 077	886 110	959 868	877 335
Acc – Scen1 total	868 455	<–	<–	<–
Scen1 – Cont1	375 476	<–	<–	<–
Scen1 – Cont2	331 156	<–	<–	<–
Scen1 – Buffers	161 823	<–	<–	<–
Acc – Scen2 total	32 622	17 655	91 413	8 880
Scen2 – Cont1	8 180	4 842	25 101	2 422
Scen2 – Cont2	3 727	1 939	9 868	959
Scen2 – Buffers	20 715	10 874	56 444	5 499
Alloc Mem – total [B]	7 838 184	7 834 560	7 841 160	7 834 152
Alloc Mem – Cont1 [B]	1 147 212	<–	<–	<–
Alloc Mem – Cont2 [B]	6 685 932	<–	<–	<–
Alloc Mem – Buffers [B]	5 040	1 416	8 016	1 008
Energy – total [nJ]	4 212 519	4 178 505	4 328 012	4 159 965
Energy Gain [%]	70	69,7	70,4	69,6

Table 4.22: Array-structured Trie Implementation – Transformation Combination – Small Dictionary

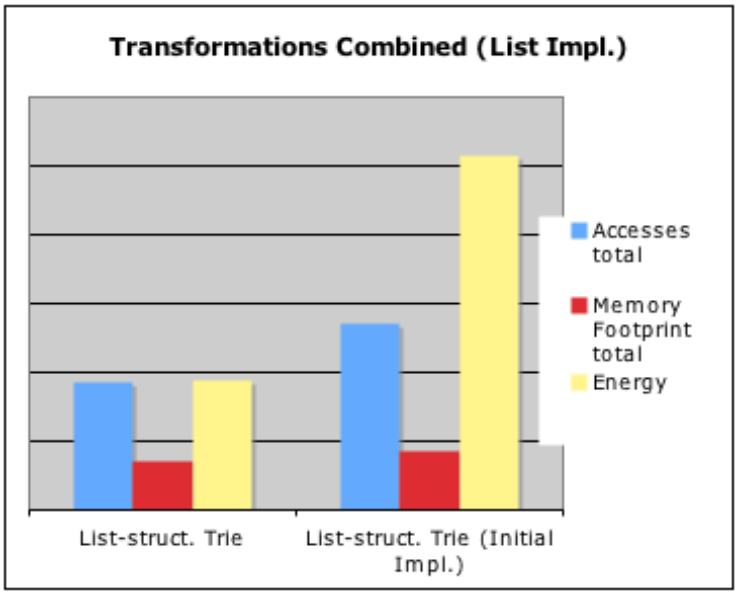

Figure 4.10: Transformations combined – List-structured Trie Compared, Text: spiegel, Small Dictionary

Chapter 5

Conclusions and Future Work

This work shows the feasibility of ADT model transformations steered by semantic (i.e., application specific) and dynamic information in the embedded systems context. By profiling selected transformations, it is shown that the energy consumption is decreased by 70% (Chapter 4). Further, trade-off points between the two cost metrics, data accesses and memory footprint, are presented. Insight into a wide range of trade-offs is specifically valuable for embedded system designers, since this allows then to choose an optimal solution for a specific metric according to the restrictions of their hardware platform.

In this work, the catalogue of transformations introduced in [32] is extended. On the one hand, new transformations are suggested and motivated, on the other hand, known transformations are shown feasible for a different problem domain, i.e., for the spell checker application instead of a 2D-graphics game application. Both the finding of new transformations, as well as validating established transformations, points towards a significant theory of ADT model transformations.

Defining a transformation through transformation rules (Chapter 3), points towards the possibility of future automation. Automation eases the error prone tasks of manual model transformations and reduces the time-to-market of the embedded system.

Due to the character of the transformations, automatic detection is not achieved in the context of this work. As shown in Chapter 2, transformations should be motivated through application and domain specific knowledge, thus making automatic detection difficult and requiring the help of the designer.

Additionally to the definition of transformation rules with invariants that ensure the correct execution of a transformation, it is shown how behavior preservation can be modeled, following a snapshot-based approach. By modeling the behavior of an application in different snapshots, it is possible to examine various relevant behavior patterns, and thus show behavior preservation at the modeling level.

Future Work

The immediate future work should include the further extension of the transformation catalogue and formalization of the transformations, in order to gain more insight. Also

a thorough definition of the ADT metamodel is needed, since this is the basis for a concrete automated solution.

Long term future work should deal with the automation of the transformations. Transformations can be supported by an application, e.g., a *Transformation Browser*, that stores models in a repository. The browser should be able to display those models in different kinds of diagrams. The designer can choose a transformation to apply out of the set of transformations in the repository.

A useful and necessary expansion of the *Transformation Browser* is a code generation engine that produces code already suited for further embedded systems optimizations.

A different branch of future work should involve the formalization of application specific knowledge. Here, a possibility is to use profiling information as application specific knowledge that augments the models and helps the designer detect transformation opportunities. For this to be applicable, the current profiling tools need to be calibrated to the informational needs of this domain. With good visualization, the profiling information can be a useful addition to the *Transformation Browser's* functionality.

Appendix A

Definitions and Acronyms

- ADT – Abstract Data Type
  'A data type for which only the properties of the data and the operations to be performed on the data are specified, without concern for how the data will be represented or how the operations will be implemented.' [1]
- CDS – Concrete Data Structure
  *'A **concrete data structure** (CDS) specifies a specific way of storing a collection of objects in memory. Examples are: Array, List, Tree.'*[30]
- Dynamic Application
  'An application whose behavior is unpredictable at design time and depends on the input at run-time (in time and value).' [30]
- Static Application
  'An application whose behavior is independent of the run-time context.' [30]
- Use Case
  *'**Use cases** are a way to capture system functionality and requirements in UML. [..] Use cases represent distinct pieces of functionality for a system, a component, or even a class.'* [24]
- Model
  *'A **model** is a description of (part of) a system written in a well-defined language.'* [15]
- Well-defined language
  *'A **well-defined language** is a language with well-defined form (syntax), and meaning (semantics), which is suitable for automated interpretation by a computer.'* [15]
- Transformation Definition
  *'A **transformation definition** is a set of transformation rules that together describe how a model in the source language can be transformed into a model in the target language.'*[15]

- Transformation Rule
  *'A **transformation rule** is a description of how one or more constructs in the source language can be transformed into one ore more constructs in the target language.'* [15]

Bibliography

[1] Alain Abran, Pierre Bourque, Robert Dupuis, and James W. Moore, editors. *Guide to the Software Engineering Body of Knowledge - SWEBOK*. IEEE Press, Piscataway, NJ, USA, 2001.

[2] Jun-Ichi Aoe, Katsushi Morimoto, and Takashi Sato. An Efficient Implementation of Trie Structures. *Software — Practice and Experience*, 22(9):695–721, 1992.

[3] David Atienza, Jose M. Mendias, Stylianos Mamagkakis, Dimitrios Soudris, and Francky Catthoor. Systematic dynamic memory management design methodology for reduced memory footprint. *ACM Trans. Des. Autom. Electron. Syst.*, 11(2):465–489, 2006.

[4] Alexandros Bartzas, Stylianos Mamagkakis, Georgios Pouiklis, David Atienza, Francky Catthoor, Dimitrios Soudris, and Antonios Thanailakis. Dynamic data type refinement methodology for systematic performance-energy design exploration of network applications. In *DATE '06: Proceedings of the conference on Design, automation and test in Europe*, pages 740–745. European Design and Automation Association, 2006.

[5] Jean-Michel Bruel, Brian Henderson-Sellers, Franck Barbier, Annig Le Parc, and Robert B. France. Improving the UML Metamodel to Rigorously Specify Aggregation and Composition. In *OOIS*, pages 5–14, 2001.

[6] Francky Catthoor, Eddy de Greef, and Sven Suytack. *Custom Memory Management Methodology: Exploration of Memory Organisation for Embedded Multimedia System Design*. Kluwer Academic Publishers, Norwell, MA, USA, 1998.

[7] Krzysztof Czarnecki and Simon Helsen. Classification of Model Transformation Approaches. In *OOPSLA 2003 Workshop on Generative Techniques in the context of Model Driven Architecture*, October 2003.

[8] Edgar G. Daylight, David Atienza, Arnout Vandecappelle, Francky Catthoor, and Jose M. Mendias. Memory-access-aware data structure transformations for embedded software with dynamic data accesses. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 12(3):269– 280, March 2004.

[9] Bruce Powel Douglass. *Real Time UML: Advances in the UML for Real-Time Systems (3rd Edition)*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2004.

[10] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, June 1999.

[11] E. Fredkin. Trie memory. *Communications of the ACM*, 3(9):490–499, 1960.

[12] Tony Givargis, Frank Vahid, and Jorg Henkel. System-level exploration for pareto-optimal configurations in parameterized systems-on-a-chip. In *ICCAD '01: Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*, pages 25–30, Piscataway, NJ, USA, 2001. IEEE Press.

[13] Pieter Van Gorp, Hans Stenten, Tom Mens, and Serge Demeyer. Enabling and Using the UML for Model-Driven Refactoring. In Stphane Ducasse Serge Demeyer and Kim Mens, editors, *Proceedings WOOR'03 (ECOOP'03 Workshop on Object-Oriented Re-engineering)*, pages 37–40. Universiteit Antwerpen, July 2003.

[14] Brian Kelk. Uk english wordlist with frequency classification, version 1.01, 2003. `http://www.bckelk.uklinux.net/words/wlist.zip`.

[15] Anneke G. Kleppe, Jos Warmer, and Wim Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[16] Marc Leeman and David Atienza Alonso. Intermediate variable elimination in a global context for a 3D multimedia application. In *Multimedia and Expo, 2003. ICME '03. Proceedings. 2003 International Conference on*, 2003.

[17] Quan Long, Zhiming Liu, Xiaoshan Li, and He Jifeng. Consistent Code Generation from UML Models. In *ASWEC '05: Proceedings of the 2005 Australian conference on Software Engineering*, pages 23–30, Washington, DC, USA, 2005. IEEE Computer Society.

[18] Tom Mens and Pieter Van Gorp. A taxonomy of model transformation. In *Proc. Int'l Workshop on Graph and Model Transformation*, 2005.

[19] Wolfgang Mueller, Yves Vanderperren, and Wim Dehaene. UML and Model-Driven Development for SoC. In *DATE 06 Tutorial Notes*, 2006.

[20] James Noble and Charles Weir. *Small memory software: patterns for systems with limited memory*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[21] OMG Object Management Group. Ocl 2 specification, version 2.0, June 2005.

[22] William F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, Urbana-Champaign, IL, USA, 1992.

[23] A. Papanikolaou, M. Miranda, F. Catthoor, H. Corporaal, H. De Man, D. De Roest, M. Stucchi, and K. Maex. Global interconnect trade-off for technology over memory modules to application level: case study. In *SLIP '03: Proceedings of the 2003 international workshop on System-level interconnect prediction, ACM press*, 2003.

[24] Dan Pilone and Neil Pitman. *UML 2.0 in a Nutshell*. O'Reilly Media, Inc., first edition edition, 2005.

[25] Christophe Poucet, David Atienza, and Francky Catthoor. Template-Based Semi-Automatic Profiling of Multimedia Applications. In *Proceedings of the International Conference on Multimedia and Expo (ICME 2006)*, Toronto, Canada, July 2006. IEEE Computer, IEEE Signal Processing, IEEE System and IEEE Communications Society.

[26] Donald Bradley Roberts. *Practical Analysis for Refactoring*. PhD thesis, University of Illinois, 1999.

[27] M. Saksena, R. B. France, and M. M. Larrondo-Petrie. A characterization of aggregation. *International Journal of Computer Systems Science and Engineering*, 14(6):363–372, 1999.

[28] Bernhard Schaetz, Alexander Pretschner, Franz Huber, and Jan Philipps. Model-Based Development of Embedded Systems. In *OOIS '02: Proceedings of the Workshops on Advances in Object-Oriented Information Systems*, pages 298–312, London, UK, 2002. Springer-Verlag.

[29] Gerson Sunye, Damien Pollet, Yves Le Traon, and Jean-Marc Jezequel. Refactoring UML Models. In *UML '01: Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools*, pages 134–148, London, UK, 2001. Springer-Verlag.

[30] Marijn Temmerman. The ADT Design Space. Internal technical report, IMEC vzw., November 2006.

[31] Marijn Temmerman. Towards Desing-Space Exploration of Data Structures from the ADT-Modeling Level. *Submitted to IJES journal*, 2006.

[32] Marijn Temmerman, Edgar Daylight, Franky Catthoor, Serge Demeyer, and Tom Dhaene. Moving up to the modeling level for the transformation of data structures in embedded multimedia applications. In *Proceedings SAMOS'05 (Fifth meeting of the Embedded Computer Systems: Architectures, MOdeling, and Simulation)*, Lecture Notes on Computer Science. Springer-Verlag, 2005.

[33] Marijn Temmerman, Edgar Daylight, Franky Catthoor, Serge Demeyer, and Tom Dhaene. Moving Up to the Modeling Level for the Transformation of Data Structures in Embedded Multimedia Applications. *Submitted to Journal of Systems Architecture*, 2006.

[34] J. Warmer and Anneke Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison Wesley, second edition edition, 2003.

[35] Sven Wuytack, Francky Catthoor, and Hugo De Man. Transforming set data types to power optimal data structures. In *ISLPED '95: Proceedings of the 1995 international symposium on Low power design*, pages 51–56, New York, NY, USA, 1995. ACM Press.