Bachelor Thesis

One-way Model Transformation

On structural and semantical one-way transformations for constraint-based models, exemplified in the context of roadmapping.

Prof. Dr. Matthias Tichy
Examiner
A. Breckel and J. Pietron
Supervisors

Florian Sihler
November, Ulm University
Institute of Software Engineering and Programming Languages
Bachelor Thesis in Computer Science
Date of Submission: 2021-11-30

Examiner
• Prof. Dr. Matthias Tichy

Supervisors
• Dipl.-Inf. Alexander Breckel
• M. Sc. Jakob Pietron
Abstract

Today’s industry depends heavily on the collaboration of multiple businesses that have to exchange a lot of information. In combination with the recent trend of technology roadmapping and model-driven development, this leads to the exchange of very large models. However, the information exchangeable is governed by competition laws, and the businesses’ own interest of not revealing their intellectual property. Manually ensuring shared models do not contain any “content worth protection” for every exchange is a tedious and error-prone process.

Therefore, this bachelor thesis suggests an automated one-way transformation approach to remove content worth protecting from a given model. With it, we propose two core artifacts: i) An obfuscation program with a formal catalog of applicable one-way model transformations, and ii) a proof of concept implementation of this obfuscation program in the context of the modeling tool IRIS.

We collect a total of sixteen transformation categories and analyze them according to a set of predefined properties and constraints. The proof of concept implementation underlines their effectiveness and reveals areas in which there exists a need for further research.

Florian Sihler
November 30, 2021

Acknowledgements

This bachelor thesis was made possible by the support and work of many people. First of all, I am extremely grateful to my supervisors Alexander Breckel and Jakob Pietron, who provided insightful feedback while trying to stop me whenever I started losing myself in details. Furthermore, I extend my sincere thanks to Prof. Dr. Matthias Tichy for allowing me to write this thesis and for all the other contributors of the IRIS-project, laying the groundwork for the proof of concept implementation.

This thesis makes use of the Font Awesome icon toolkit and font framework(1) licensed under the SIL Open Font License, version 1.1.(2)

IRIS is developed in the GENIAL!-project (reference number: 16ES0875) partly funded by the BMBF within the research programme ICT 2020 and the DFG — 453895475. Screenshots of IRIS and accompanying text makes us of icons by the following authors and sets:

Yusuke Kamiyamane
CC-BY Attribution 3.0 License
https://p.yusukekamiyamane.com

Silk Icon Set
CC-BY Attribution 3.0 License
http://www.famfamfam.com/lab/icons/silk

FatCow
CC-BY Attribution 3.0 United States License
https://www.fatcow.com/free-icons

(1) https://fontawesome.com/ (11/08/21)
(2) https://opensource.org/licenses/OFL-1.1 (11/08/21)
BMBF: German Federal Ministry of Education and Research.
DFG: German Research Foundation.
Short Overview

1 Introduction
   1.1 Motivation ................................................. 1
   1.2 Problem Statement ........................................ 2
   1.3 Research Questions ....................................... 6
   1.4 Research Method ......................................... 6
   1.5 Overview ................................................. 8

2 Background Information and Related Work .......................................... 9
   2.1 Roadmapping in IRIS ....................................... 9
   2.2 Formalization ............................................. 16
   2.3 Model Transformation ..................................... 21
   2.4 Related Work ........................................... 22

3 One-Way Transformation ................................................. 27
   3.1 Overview ................................................. 27
   3.2 Classification ............................................ 34
   3.3 Transformation Catalog ................................... 37
   3.4 Limitations ............................................. 63
   3.5 Execution Rules .......................................... 65

4 Proof of Concept Implementation ................................................. 71
   4.1 Preparations ............................................. 71
   4.2 Implementation Scope ..................................... 73
   4.3 User Interface ........................................... 75
   4.4 Challenges .............................................. 78
   4.5 Evaluation ............................................... 80

5 Conclusion and Future Work ................................................. 87
   5.1 Conclusion ................................................ 87
   5.2 Problems and Limitations .................................. 88
   5.3 Future Work ............................................... 89

A Sources
   A.1 Bibliography .............................................. 91

B Pseudocode Reference ................................................. 101
   B.1 General Rules ............................................ 101
   B.2 Reference ................................................ 101

C Further Lists ................................................. 105
   C.1 List of Figures ........................................... 105
   C.2 List of Tables ............................................ 106
   C.3 List of Examples ......................................... 106
   C.4 List of Digital Attachments ............................... 106
Detailed Contents

1 Introduction

1.1 Motivation ......................................................... 1
  1.1.1 Used Framework ..................................... 2
  1.1.2 Disclaimer ................................................... 2
1.2 Problem Statement ............................................ 2
  1.2.1 Model Notation ......................................... 2
  1.2.2 Behavioral Equivalence ................................. 3
  1.2.3 Problem Specification ................................. 4
  1.2.4 Proof of Concept Requirements ...................... 5
1.3 Research Questions ........................................... 6
1.4 Research Method ............................................... 6
  1.4.1 Analysis Phase .......................................... 6
  1.4.2 Design Phase ........................................... 7
    Desired results ........................................... 7
  1.4.3 Evaluation Phase ....................................... 7
  1.4.4 Diffusion Phase ........................................ 8
1.5 Overview ....................................................... 8

2 Background Information and Related Work ........................................... 9

2.1 Roadmapping in IRIS ........................................... 9
  2.1.1 Structure and Hierarchy ................................. 10
    Solution Spaces .......................................... 10
    Hierarchies ............................................. 11
  2.1.2 Model Elements ......................................... 12
  2.1.3 Expression Language ................................... 13
    Intervals and Uncertainty ................................. 13
    Ternary Logic .......................................... 14
    Identifiers ............................................. 14
    Dates and Durations ..................................... 14
  2.1.4 Solving .................................................... 15
    Impossible Bounds ...................................... 15
    Overwriting Behavior .................................. 16
2.2 Formalization .................................................. 16
  2.2.1 Structure and Hierarchy ................................. 17
    Normalized Model ....................................... 18
  2.2.2 Formal Language ......................................... 18
    Syntactic Elements ................................... 19
    Known and Unknown Functions .......................... 19
    Known and Unknown References ....................... 19
    Requirements ........................................... 19
  2.2.3 Formal Solver ............................................ 20
    Formal Behavioral Equivalence ......................... 20
    Tolerable Behavioral Equivalence ..................... 20
  2.2.4 External Factors ......................................... 21
2.3 Model Transformation ......................................... 21
2.4 Related Work .................................................. 22
  2.4.1 Model Transformation .................................. 22
3 One-Way Transformation

3.1 Overview ............................................. 27
  3.1.1 First Phase – Normalization ......................... 28
    ▶ Important Aspects .................................. 28
    ▶ Implicit Assumptions .............................. 29
    ▶ Flattening and Constraint Generation .......... 29
    ▶ Tracing ............................................ 30
  3.1.2 Second Phase – Preparation ....................... 31
    ▶ Preconditions .................................... 31
    ▶ Postconditions .................................. 32
  3.1.3 Third Phase – Obfuscation ......................... 32
    ▶ Postconditions .................................. 33
  3.1.4 Fourth Phase – Verification ....................... 34
    ▶ Postconditions .................................. 34

3.2 Classification ........................................ 34
  3.2.1 Overall Allocation .............................. 35
  3.2.2 Transformation Properties ....................... 35

3.3 Transformation Catalog ............................. 37
  3.3.1 Overview ....................................... 37
  3.3.2 General Transformations ......................... 39
  3.3.3 Disguise of Computing Paths .................... 42
  3.3.4 Disguise of the Result ........................ 53
  3.3.5 Disguise of the Structure ....................... 56

3.4 Limitations ........................................... 63
  3.4.1 Possible Adversaries and Legal Issues .......... 63
  3.4.2 Theoretical Limitations ........................ 64
  3.4.3 Limited Transformations and Formalization .... 64
  3.4.4 Threats to Validity ............................ 65
    ▶ Missing Properties .............................. 65
    ▶ Generalization ................................ 65

3.5 Execution Rules ..................................... 65
  3.5.1 Subphases ...................................... 66
  3.5.2 Outer Subphase ................................ 66
  3.5.3 Inner Subphase ................................ 67
  3.5.4 Structure Subphase ............................ 68
  3.5.5 Cleanup Subphase .............................. 68

4 Proof of Concept Implementation ....................... 71
  4.1 Preparations ...................................... 71
    4.1.1 The Solver of IRIS .............................. 72
    4.1.2 Specific Implicit Assumptions .................... 73
  4.2 Implementation Scope .............................. 73
    4.2.1 Adaption of Selected Transformations .......... 73
      ▶ Essential Adaptions ............................ 73
    4.2.2 Block-Based Selection Mechanism ............... 75
    4.2.3 Unsupported Elements ........................ 75
  4.3 User Interface ...................................... 75
    4.3.1 Exporting the Obfuscated Model ................ 76
  4.4 Challenges .......................................... 78
    4.4.1 Recovering Lost Information .................... 78
    4.4.2 Importing Models .............................. 78
4.4.3 Increasing Usability ......................................... 79
4.4.4 Solution Alternatives ........................................ 80
4.4.5 Overwriting Implicit Assumptions ......................... 80
4.5 Evaluation ......................................................... 80
   4.5.1 Study Design .................................................. 80
   4.5.2 Correctness ................................................... 81
      ▷ Test Coverage .................................................. 81
   4.5.3 Benchmarking .................................................. 82
      ▷ EFuse Simplified ................................................. 82
      ▷ Sensors .......................................................... 83
      ▷ Other Models ..................................................... 84
   4.5.4 Threats to validity ............................................. 84
      ▷ Reliability and Reproducibility ............................ 84
      ▷ Internal Validity .................................................. 84
      ▷ External Validity .................................................. 85

5 Conclusion and Future Work .................................. 87
   5.1 Conclusion ......................................................... 87
   5.2 Problems and Limitations ..................................... 88
      5.2.1 Restrictions of IRIS ......................................... 88
      5.2.2 Limitations regarding the Obfuscation ................. 89
   5.3 Future Work ..................................................... 89

A Sources .................................................................... 91
   A.1 Bibliography ....................................................... 91

B Pseudocode Reference ............................................ 101
   B.1 General Rules ...................................................... 101
   B.2 Reference .......................................................... 101
      B.2.1 Tree Related ................................................. 101
      B.2.2 Modifying Data Structures ............................ 102
      B.2.3 Functions ..................................................... 102
      B.2.4 Values .......................................................... 102
      B.2.5 Utility ............................................................ 103

C Further Lists .......................................................... 105
   C.1 List of Figures ....................................................... 105
   C.2 List of Tables ....................................................... 106
   C.3 List of Examples .................................................... 106
   C.4 List of Digital Attachments ................................. 106
1 Introduction

This chapter aims to enrich the reader with a brief understanding of the problem, its importance in the real world, and the formal approach used to produce the solution presented afterwards. Among others, the following three questions are answered: i) What is the major challenge to be addressed, ii) what are the steps to solve it, and iii) what is to be expected of the thesis and the other chapters?

1.1 Motivation

Value chains are one of the fundamental concepts in modern businesses and often require extensive tool support, for example through technology roadmaps. Those roadmaps enable collaboration and help to identify industry needs by operating more holistically and granting invaluable insights for the business [43].

Technology roadmaps produce models representing and forecasting the development and structure of innovation [70]. Because of their omnipresence, there are a lot of different formats and types (cf. [67, 44]). Nonetheless, their observable behavior — like the time-dependent availability of an innovation — can be expressed by a set of constraints that has to be solved to determine the state of a technology or a product at a certain point in time.

As many innovations require the collaboration of multiple parties, there are many reasons against distributing the complete model to everyone involved. Those include a) simplifications or complexity reductions so that the receiving party can grasp the model more easily from its perspective, b) the removal of content that is worthy of protection such as trade secrets and intellectual property (IP), or c) staying compliant with various competition laws [18]. As a result, every party only owns and knows pieces of one full model, which may never be viewed or stored as a whole by anyone (e.g., due to compliance reasons).

Nevertheless, the parties have to exchange information so they can work together. As a simple example, consider a fuse manufacturer that needs information about the expected and critical currents (which may change over time due to innovations) and, in return, a customer who has to know the tolerances of the fuse proposed.

While that information can be extracted by hand, it is a tedious and impractical process. Moreover, with a lot of parties, communication, and changes, the manual (re-)creation of sub-models for specific partners is costly and error-prone, increasing the risk of leaking unwanted or unauthorized information [31]. Accordingly, there is the need for an automatic transformation that processes a model, so it only contains the information desired for one party [15, p. 19].
1.2 Problem Statement

1.1.1 Used Framework

The GENIAL!-project\(^{(1)}\) aims to improve the interaction of multiple parties in the automotive industry. Its graphical modeling tool \textsc{Iris} is used as the basis for a proof of concept implementation of the one-way model transformation approach proposed in this thesis (cf. Section 3.3). Even though, the transformations are characterized and formalized independently of any concrete model format (e.g., a technology roadmap), their applicability and effectiveness is exemplified by their integration into \textsc{Iris} [15].

1.1.2 Disclaimer

Due to the ubiquity of technology roadmaps and model-driven management in general, there is a wide variety of possible approaches to model transformation. Furthermore, there are various preexisting technologies.

Broadly speaking, even some “conventional” model-to-code transformations like those performed by compilers (cf. [2, 29]) can be classified as one-way transformations simply because they obscure the program’s representation. However, they operate on different levels of abstraction and are predominantly used in model-driven architectures (MDA, cf. [74]).

Other model-to-model transformations focus on bidirectional translations to synchronize changes in multiple \textit{UML}-Diagrams of the same system (e.g., block diagrams to class diagrams [45]).

We are not aware of any research on transformations in the exact same context of the presented scenario (cf. Section 1.2 and Section 2.4). However, there are cases in which there can never be a sufficient set of transformations to assure the complete removal of any vital information. This limitation is mostly due to i) the informality of the underlying rules (e.g., competition laws [18]) which would have to be fulfilled by the model transformation process, and ii) families of functions that can not be fully obfuscated [64]. We elaborate these limitations in Section 3.4 and Section 5.2.

Scope of this Bachelor Thesis

While the motivation focuses on the domain of roadmapping technology, the problem applies to \textit{IP}-preserving and compliant model exchanges between businesses in general. Therefore, the focus of this bachelor thesis lies in elaborating and justifying a more general approach that applies to problems beyond the scope presented in the introduction.

1.2 Problem Statement

This section specifies the definition of \textit{models} used in this document and coins the term \textit{behavioral equivalence} which is key to the precise problem formulation and its accompanying constraints. Furthermore, it lists a concrete set of requirements to be met by the transformations and their implementation in \textsc{Iris}. For additional definitions and formalizations see Section 2.2.

1.2.1 Model Notation

In the scope of this thesis, we describe a model as a set of \textit{model elements}. All model elements represented by an unique identifier. For now, we consider only one type of model element, the “formula model element”. It assigns a formula to its unique identifier.
The formula is to be evaluated by an arbitrary set of predefined and fixed syntax- and semantic rules. For the problem statement, we allow units denoted in accordance to the SI system.

The model in Figure 3/A showcases a simple “formal” model \( A \) that contains only two model elements with the unique identifiers \( a \) and \( b \). The formula attached to \( a \) is \( 2 + 5 \), while the formula for \( b \) is \( 3 \text{kg} \). The set containing all the unique identifiers present in the model \( A \) is denoted by \( E_A \) and is \( E_A = \{ a, b \} \).

Refer to Section 2.2 for a more in-depth overview of the formalization.

### 1.2.2 Behavioral Equivalence

This thesis, and especially the following specification, requires a mechanism to compare the behavior of two or more models — which allows making statements about a model and its one-way transformed counterpart. Therefore, we need a definition of what is part of the behavior of a model and what is not.

The authors Yücesan and Schruben present various definitions for sensible equivalence constructs based on simulation graph models [88]. For this thesis, we define the behavior of a model resembling their definition of the “input/output behavior”. This behavior includes the values to which model elements evaluate, the availability of specific components, and more. In short, every behavior exhibited by the model that can be perceived by the user for a given input.

**Consider the model**

\[
M = \{ a : 2 + 5, b : 3 \text{kg} \}
\]

which is the same as the one found in Figure 3/A. Using units, the output behavior of this model is the solution 5 for the model element \( a \) and 3 kg for the model element \( b \). As this model does not have any input, the output behavior is static and does not change.

However, for the concept of “behavioral equivalence” we do not just compare the input/output behavior of models when comparing a model with its one-way transformed variant. If we would only compare their behavior, the comparison would fail as soon as one model contains an element that the other one does not — which is almost certainly the case because we want to remove information.

Therefore, we restrict the term of behavioral equivalence to model elements both models have in common. We write the set of common model elements as the intersection of their elements: \( E_A \cap E_B \). Furthermore, we allow all external inputs (like time) to be restricted by a certain domain. In other words, the definition of behavioral equivalence reads as follows:

\[
\forall t \in [t_s, t_e] \forall e \in E_A \cap E_B : e_A(t) = e_B(t).
\]

We use \( e_A \) and \( e_B \) to identify elements that have the same identifier but stem from two different models. Therefore, \( e_A \) refers to the model element \( e \) from the model \( A \) and \( e_B \) refers to the model element \( e \) from the model \( B \). With \( e_A(t) \) and \( e_B(t) \) we denote the evaluation of the given element at a concrete point in time \( t \), with \( t \) being restricted by the interval \( [t_s, t_e] \).

In other words: we compare the input/output behavior of all model elements which two models have in common by comparing the output for every possible input (which may be restricted). The behavioral equivalence is trivially fulfilled for two models that do have no elements in common. The following Example 4/A presents a case in which two models are considered to be not “behavioral equivalent”.

---

**SI**: The international System of Units.

\[
\begin{align*}
A &= \{ 
  a : 2 + 5, \\
b : 3 \text{kg}
\} \\
\end{align*}
\]

**Figure 3/A**: A simple model with elements \( E_A = \{ a, b \} \).

**Model**

Set of model elements.

**Model Element**

Uniquely identified element of a model.

**\( E_M \)**

Set of unique identifiers in model \( M \).

**Example 3/A**: Behavior of a model.

**Behavioral Equivalence**: All variables in common evaluate to the same value for all external factors.

(2) The property of common elements is weakened later to allow for an bijective identifier mapping, so that the unique identifiers can be altered.
Consider the two models $A$ and $B$:

$$A = \{ a : 2 + 5, \ b : 3 \text{kg} \}$$

$$B = \{ a : 3 \}$$

With $E_A = \{ a, b \}$ and $E_B = \{ a \}$ this results in $E_A \cap E_B = \{ a \}$. Yet, $a_A$ (being $2 + 5$ or in other words $7$) does not equal $a_B$ (being $3$) for any time-interval $[t_s, t_e] \neq \emptyset$. Therefore, the models $A$ and $B$ are considered to be not “behavioral equivalent” in the terms of this thesis.

This definition has various beneficial properties, which are explored further in the rest of this thesis.

This thesis does not consider models that do not have any output behavior as they do not have anything that can be obfuscated reasonably (i.e., no information has to be retained). However, this thesis covers models that have no input (although the definition expects the existence of time) by modifying their behavioral equivalence so that it does not compare common elements for all points in time or all external factors but only once for their static evaluation result. This special treatment does not affect any of the following definitions. However, it excludes some transformations that deal with the domains of external factors. If there are no external factors, transformations handling them are redundant.

### 1.2.3 Problem Specification

Given a model $A$ with its set of model elements $E_A$ and a selection of those elements $S \subseteq E_A$, the problem lies in creating another model $B$ which satisfies the four following constraints (cf. Figure 4/A on the left side) that we consider important to ensure that $B$ no longer contains any content worth protecting while keeping the behavior of the selection intact:

- **C/1** $B$ has to be self-contained, without any reference to any $a \in E_A$.
  
  Implementing this is trivial by constructing $B$ as a complete copy of the original model $A$. This new copy can be self-contained just like $A$ and does not need any back-reference as it contains the same elements that can refer to their counterparts in $B$.

- **C/2** $B$ contains all of the elements in $S$ and none of those in $E_A \setminus S$.
  
  This second constraint does not add a great deal of complexity. Given the selection $S$ we restrict the copy to all model elements $e \in S$, replace all references to the model $A$ with placeholders, and produce a new model $B$ which is self-contained and includes exactly the elements of the selection.

- **C/3** $B$ may allow no inference on any model element $e \in E_A \setminus S$.
  
  Like with the previous two constraints, satisfying the third constraint is not a problem. We can make use of the previously described process and additionally remove any reference to elements, which are not included in the selection. The resulting model $B$ is self-contained, includes all model elements $e \in S$, and lacks any reference or information about any model element $e \in E_A \setminus S$.

- **C/4** The models $A$ and $B$ have to be behavioral equivalent.
This final constraint elevates the problem from “almost trivial” to (probably) impossible realms.\(^1\) We have to retain aspects from A which are not selected but relevant for the behavior of selected elements without including any critical information about their origin and without embedding them directly, while keeping B self-contained.

Nevertheless, solving this problem remains trivial for \(S = \emptyset\) and \(S = \emptyset\). If all model elements from A are selected, B can simply be a complete copy of A and if no element is selected, the resulting model can be empty. Therefore, the main focus of this thesis lies on cases where \(S \neq \emptyset\) and \(S \neq \emptyset\).

Instead of just “B” we write “\(B_{A,S}\)” to represent a model B created from another model A with the selection \(S\). If the model “\(B_{A,S}\)” further satisfies all four constraint, we call it a “huskied model” or plainly a “target” or “output model”. We use the name Huskied Model in the terms of a veiled or blurred model.

Considering C/3, we focus on obfuscating the model alongside all contained formulas but abstract away from individual data most of the time. That is, we do not consider the scramble of employee names in formulas or the exchange of Internet Protocol addresses because of the amount of existing research on this topic in the area of PPDM. We cover this briefly in Section 2.4 and describe it as potential further work in Section 5.3.

### 1.2.4 Proof of Concept Requirements

When dealing with the obfuscation of models in IRIS — or any concrete modeling environment — it should be possible to select elements and obfuscate them automatically. To make working with the obfuscation process practical, interacting with the transformation mechanism should be transparent to the user requiring little or no configuration.\(^4\) In the context of the proof of concept implementation in IRIS, this results in two major (R/1, R/2) and one optional (R/3) requirement. They are described in the following:

**R/1. Transformation**

The resulting model \(B_{A,S}\) must fulfill the four constraints from Subsection 1.2.3. In other words, the transparent and automated obfuscation process should satisfy all four constraints. We describe a formal approach to this problem in Chapter 3 and discuss its application in praxis in Chapter 4.

**R/2. Integration**

Huskied models should follow the same structure and syntax rules as the corresponding source model, so they can be used with the same features. For the proof of concept implementation, this requires \(B_{A,S}\) to be usable with the existing features of the IRIS-solver (i.e., it has to be an ordinary IRIS-model).

Therefore, it should be possible i) to create other components in the imported model, and ii) to create another huskied model from those components and the imported model. We address this again in Chapter 4.

**R/3. Validation**

For any model to be exported, IRIS should present a preview of the result so that the modeling user or reviewer can check the huskied model. Additionally, the validation allows for easy manual changes to \(B_{A,S}\). We discuss this further in Chapter 4.
1.3 Research Questions

While the main question of this work is how to solve the problem statement defined in Section 1.2, we enrich the answer with a set of research questions.

RQ1: What are existing techniques to ensure compliant collaboration?

There is a wide variety of preexisting work. We address this by the related work section in the “Background” chapter and reference further work alongside the description of our approach in Chapter 3.

RQ2: What are specific requirements for the target model?

While the baseline requirements are covered by the competition laws of the countries the collaborating companies reside in (cf. Subsection 3.4.1), we use the requirements listed in Subsection 1.2.4 for this thesis.

RQ3: What are necessary and adequate transformations to produce a sufficiently transformed target model (as defined by RQ2 and Section 1.2)?

The main part of this question is essentially answered by the catalog presented in Section 3.3. In more detail, we formulate the following sub-questions:

i) How flexible are those methods in real-world use?

We answer this research question alongside the required adaptations for the proof of concept implementation in Chapter 4.

ii) How can the compliance conformity and the correctness of such methods be verified?

While we argue for the impossibility of a general automated verification of compliance conformity in Subsection 3.4.1, the behavioral equivalence can be checked by an algorithm (cf. Note 75/8).

RQ4: How to formalize and describe transformations?

We present our formalization in Section 2.2 and describe them using either pseudocode or a abstract notation in Section 3.3.

1.4 Research Method

There are a lot of different approaches to research design. In “Memorandum on design-oriented information systems research” the authors present their design-oriented approach on information systems research [66]. Because this approach aims to create artifacts in the context of their usage and their benefits for businesses, the methodology of this thesis is based on theirs. The four phases of this approach are discussed in the remainder of this section.

1.4.1 Analysis Phase

The analysis phase is about the problem identification and, therefore, primarily answered by the introduction (i.e., Chapter 1). The problem statement in Section 1.2 characterizes the problem, Section 1.3 lists the important research questions, and this section describes the problem-solving approach.

The analysis phase results in a deepened understanding of the problem alongside a set of four concrete constraints C1–C4 to be met by the transformations. The artifacts produced in the design phase are to be analyzed with the help of these four constraints.
1.4.2 Design Phase

The goal of the design phase is the creation and justification of artifacts. In the context of this bachelor thesis, this corresponds to the formalization, analysis, and sophisticated discussion of transformations, as well as the creation of a proof of concept implementation in IRIS.

The formalization created is to be found in Section 2.2. In addition to the specification of transformations, there is a need for rules guiding their execution. Those execution rules have to define i) the order in which the transformations should operate, ii) the applicability of a transformation, and iii) the end of the transformation process as a whole (e.g., how to detect the exhaustion of a transformation). These execution rules are described in Section 3.5.

The implementation in IRIS is the second artifact that acts as a proof of concept and therefore showcases the transformations in practice. This is especially important with R/2, as this requirement demands additional features (e.g., the selection of elements and the export of the given model) expected to be “given” in the formalization.

As IRIS is implemented in TypeScript, the implementation language is the same. Furthermore, the implementation itself is verified by an extensive set of tests (cf. Note 72/A).

Desired results

While the theoretical plan of this phase might appear linear, the execution is not. New transformations require additional formalisms. The implementation in IRIS reveals more challenges that, in turn, require additional transformations or methods.

The results of this phase are: i) a collection of formalizations that serve as a foundation for the analysis, ii) various transformations, analyzed by the defined formalisms, iii) a set of execution rules defining the applicability and order of the transformations, and iv) a proof of concept implementation in IRIS. They are explained in Chapter 3 and Chapter 4.

1.4.3 Evaluation Phase

This evaluation phase aims to verify the generated artifacts — transformations alongside their execution rules and the proof of concept implementation. As the artifacts differ in type, so do their verification strategies. Please note that, while formal and mathematical proofs are the ultimate goal, they turn out to be too hard or not possible in the given time frame in some cases therefore exceeding the scope of this thesis (cf. Section 3.4).

- The transformations are partially validated by the proof of concept implementation, yet their correctness has to be assured separately. For preexisting methods, the proof remains largely unchanged from the one presented in the original source. Newly created or modified transformations are analyzed with the help of the formalization. The execution rules are based on heuristics and therefore hard to prove. However, they are analyzed and discussed in an idealized scenario in Section 3.5.

- The proof of concept implementation is verified by the extensive set of unit tests and a collection of models. The formal correctness is inferred from the underlying transformations.

The result of the evaluation phase is a verification of the artifacts created in the design phase. It is discussed in Section 4.5 and Chapter 3.
1.4.4 Diffusion Phase

The diffusion phase deals with the dissemination of the findings amongst the target audience. This dissemination is handled by this bachelor thesis and the proof of concept implementation in IRIS, as both of them may be subject to further research and innovation in the context of the GENIAL-project (cf. Section 5.3).

1.5 Overview

At last, this section introduces the reader to the other chapters in this thesis.

Background Information and Related Work. Before the explanation of the one-way model transformation approach, Chapter 2 equips the reader with knowledge of various fundamentals for the following chapters. In addition, the chapter gives an overview of existing research in related areas.

One-Way Transformation. Based on the groundwork of Chapter 2, Chapter 3 presents a detailed description and discussion of the proposed model transformation approach, that has to satisfy all four constraints from Subsection 1.2.3 while assisting in meeting all three requirements from Subsection 1.2.4. While this chapter focuses on a formal foundation of the approach and remains mostly theoretical, it facilitates the practical proof of concept implementation in the following chapter.

Proof of Concept Implementation. With the formal foundation from Chapter 3, Chapter 4 presents the methods used in the proof of concept implementation in the context of the IRIS modeling environment. It elaborates on the most important challenges of adapting the formal approach to the proof of concept implementation and explains countermeasures to address them.

Conclusion and Future Work. Finally, the last chapter takes the artifacts created in the previous two chapters: i) the formal foundation, and ii) the proof of concept implementation, to discuss their contribution as well as remaining open problems. Furthermore, this chapter describes possible future work in the area of one-way model transformation.
2 Background Information and Related Work

Roadmapping in IRIS, 9 • Formalization, 16 • Model Transformation, 21 • Related Work, 22

This chapter provides the reader with necessary background information on model transformation and related areas. Therefore, it introduces the usage of models in IRIS, describes how they are to be formalized, and continues with an overview of various model transformation techniques. Lastly, it covers existing research in related areas, such as information concealing, cryptography, and obfuscation.

2.1 Roadmapping in IRIS

Before we describe our formalization in Section 2.2, this section summarizes the most important features of IRIS (cf. [15]). This includes i) the hierarchical structure, ii) the modeling language, and iii) the external factor time. Knowledge of these features is important to follow the proof of concept implementation in Chapter 4. Furthermore, it helps to understand our formalization in the following Section 2.2.

Figure 9/A on the right side presents the meta-model used in IRIS. In general, IRIS represents a model as a collection of blocks, with blocks structuring the model-elements contained. All model elements — blocks, properties, constraints, requirements, KPIs, and notes — are identified by an uniquely generated identifier. Additionally, some model elements can have an “Expression” attached, that uses a domain-specific textual language which is part of IRIS (cf. Subsection 2.1.3). To reference other elements in an expression, IRIS requires all referenceable model elements — blocks, properties, and requirements — to be assigned a human-readable name. In these expressions, this name can be used instead of the uniquely generated identifier to reference those elements.(1) For now, we ignore the problem of name collisions when using this human-readable name. Besides those expressions IRIS allows for two relationships between blocks: i) a children, and ii) an interface-implementation relationship.

All model elements are explained further in Subsection 2.1.1 and Subsection 2.1.2. The domain-specific expression language is presented in Subsection 2.1.3.

Figure 10/A on the next page contains a general overview of IRIS, which presents the IRIS-frontend with an example model loaded. It serves as an overall example referenced by the following explanation.

(1) IRIS allows to reference elements by their “name” or directly by their id (cf. Subsection 2.1.3 (on page 14)).
2.1 Roadmapping in IRIS

**Figure 10A**: An Overview of IRIS. Below the toolbar at the very top, the timeslider — pictured as a road — allows changing the currently simulated time by moving the car. Thereunder are two components: a sidebar on the left showing a Gantt chart and the main model view on the right.

2.1.1 Structure and Hierarchy

The structure of a model is described by blocks, each visualized as a rectangle with a blue titlebar and the "-icon in Figure 10/A (e.g., Company, Product, and Machine A). Blocks are model elements that contain a list of other model elements — like properties and constraints — representing concepts from the modeled domain (e.g., components or technologies). Furthermore, they can contain other blocks as (direct) children. In Figure 10/A, the blocks Product and Machine are both direct children of their parent block Company.

**Solution Spaces**

Besides the children-relationship, blocks can relate to each other as interfaces and implementations. Blocks inherit all properties and requirements from their interface(s) and can override inherited properties, but not inherited requirements. For each implementation, the implemented interfaces are listed alongside the other model elements denoted by the -icon.

Properties and requirements are explained further in the subsequent Subsection 2.1.2. In a nutshell, a “property” consists of a name and an assigned expression describing its value, a “requirement” consists of a boolean expression determining if the concept represented by the block is considered “available” (cf. Note 11/A). In the context of Figure 10/A, the three blocks Machine A, Machine B, and Machine C are all implementations of the Machine interface. Each machine implementation overrides the inherited property Efficiency, yet none of them alters the inherited property Required Resources. The inherited requirement R1 from Machine can not be overridden and is implicitly part of the requirements to be fulfilled by each machine implementation.

Implementations can serve the purpose of solution alternatives if it is desired to select one specific implementation from a collection. In Figure 10/A, the solution alternatives Machine A, Machine B, and Machine C are summarized in a table to the right of the Machine interface block. All currently available implementations (i.e., those whose requirements are all fulfilled) are part of the "solution space" of an implementation: they describe the set of implementations from which one specific can be chosen. (3)

---

(2) Blocks can even be interfaces and implementations at the same time [15].

(3) IRIS allows to override the selected solution alternative manually at any time, ignoring the availability (Note 11A). This manual feature exceed the scope of this explanation but is covered by the proof of concept implementation.
In IRIS, every block has an “availability” assigned, which indicates if the abstract concept or component represented by the block is available in the current state (i.e. at the current time selected by the timeslider). Three factors determine the availability of a block. More precisely, for a block to be considered available:

1. all of its (direct) children have to be available.
2. at least one of its implementations has to be available (if the block has no implementation, this is always fulfilled).
3. all of its requirements have to evaluate to true (cf. Subsection 2.1.2).

The background color of a block's titlebar indicates its availability. In Figure 10/A, Machine A is currently available (indicated by the green background) and Machine B is no longer available (indicated by the orange background). Machine B was available once — at an earlier point in time — but now its requirement RMin is no longer fulfilled. At last, Machine C is not yet available, as indicated by a yellow background — it becomes available in the future. The other blocks like Company are always available and not colored differently.

In an expression, IRIS allows using either the name of the block or its unique identifier to reference its availability.

Another implicit aspect of a block, its replacement, is discussed in Note 11/B.

KPIs can automate the selection of a solution from the solution space using a metric. The solver prefers implementations that receive the highest value (assigned by the metric) if there is a single KPI or the highest sum of all their values if there are multiple. In Figure 10/A, Machine C scores highest on the single KPI named KPI1, yet the solver chooses Machine A as it scores highest among the available implementations. More precisely, Machine A is the only available implementation at the current time.

Besides the availability (cf. Note 11/A), all blocks are assigned a replacement. This expression evaluates to a whole number and ultimately decides which of the solution alternatives is chosen. The replacement is generated for blocks with and without solution alternatives for consistency reasons.

The implicit replacement can not be referenced directly in an expression. However, it is important to know of it, as it plays a critical part in the obfuscation of solution alternatives, as is described in Subsection 4.4.4.

Hierarchies

Blocks serve another purpose: they act as namespaces. Suppose an expression inside an element of the block Company from Figure 10/A wants to access the Efficiency of the Machine interface. In other words, we want to access the inferred Efficiency of the currently selected implementation of the Machine interface. In that case, it has to prefix the name of the block that contains Efficiency. Therefore, the reference in the expression has to read Machine.Efficiency — with the dot acting as a hierarchy separator.

Those references can be given a) absolute by starting with a block that has no parent and does not implement an interface, or b) relative to the current scope. Therefore, Efficiency in KPI1 of the block Machine refers to the Efficiency property declared in the Machine block because KPI1 exists in the same namespace as Efficiency.

For a more elaborated example, consider the requirement RMin in the block Machine A. The formula Product.Requests >= 0 references the “Requests” property of the block Product even if the block is nested inside of Company.
This relative referencing is possible because just like children gain access to the namespace of their parent, implementations gain access to the namespace(s) of the interface(s) they implement.

The following Example 12/A showcases another hierarchy with a deeper nesting level than that shown in Figure 10/A.

Even though block relationships allow to expand the accessible namespaces, IRIS employs precise and deterministic rules when resolving the targeted model element. Their discussion exceeds the scope of this explanation.

2.1.2 Model Elements

As indicated by Figure 9/A, there are six types of model elements that are of interest in this discussion: blocks, properties, constraints, requirements, KPIs, and notes. This subsection focuses on the latter five, as blocks were already explained in the previous Subsection 2.1.1. Therefore, we look at the following five model elements, prefixed by the icon that IRIS uses for them.

Properties consist of a name and a formula that describes their value. Their evaluated value for the current time is inferred by a solver that is built-in into IRIS. Its result is displayed right of the symbol.

Constraints enforce their expression, so the solver has to satisfy it. Therefore, writing \( x = 3 \) enforces \( x \) to be 3 so the constraint is not violated.

Requirements consist of a name and an expression describing a boolean condition that characterizes the availability of a block. The color of their background indicates their satisfiability.

KPIs have an expression that describes a metric which assesses competing solution alternatives as described in Subsection 2.2.1.

Notes may be used to annotate a model. They do not influence the input/output behavior but can be used as comments.

When comparing Figure 9/A with Figure 10/A it may seem inconsistent for elements like requirements or KPIs to have a name in the visual representation but none in the meta-model. The additional names are only for usability reasons. Their name or their unique identifier can not be used to reference the respective element in an expression. Therefore, they exist only to allow modeling users to give those elements a human-readable name.
2.1.3 Expression Language

The expression language of IRIS offers a wide range of syntactic elements. It can be used inside the formulas of properties, the conditions of requirements, the facts of constraints, and the metrics of KPIs as shown in Figures 9/A and 10/A. The following list contains the most important ones (cf. [15]):

- Arithmetic, relational, and boolean operators such as +, −, ×, ÷, and ∨. Furthermore, parenthesis work intuitively.
- Mathematical, boolean, and numerical constants like π, true, and 4.
- Conditionals in the form `if p then a else b`. Where p should evaluate to a boolean value. If it is true, the expression evaluates to a, if it is false, the expression evaluates to b.
- Utility functions such as sin, max, and floor.
- Identifiers, referencing other model elements (cf. Subsection /two.oldstyle./one.oldstyle./three.oldstyle).
- Intervals in the format `[a..b]` denoting a closed interval: `[a, b] ⊆ R` and representing a value range (cf. Subsection 2.1.3).
- Literals for dates like Jan2021 representing the month "January, 2021".

Besides those, IRIS allows for numerical constants to have a unit as a suffix. As an example, 600kV corresponds to 600 kilovolts.

The precedence rules employed by IRIS are similar to those found in common programming languages such as Java. Therefore, writing 3 + 2 = 4 | 4 <= 1 + 2 * 3 is the same as writing:

(`((3 + 2) = 4) | (4 <= (1 + (2 * 3)))`)

Please note, that "=" and not "==" is used to check for equality (to be more precise, "==" is not even a valid operator in IRIS).

While the presented list is incomplete, it covers a wide range of the available feature-set. We elaborate on this in the following segments.

Intervals and Uncertainty

IRIS allows to express the domain of a value using intervals. They are not to be mistaken as values themselves. Therefore, writing `X = [1..3]` is not saying `X “is”` the interval `[1, 3]` but "the value of X has to lie in the closed interval [1, 3]". Yet, all common operations on intervals are supported (cf. [15]) and defined as inclusion isotonic [63]. This means, that for intervals A1, A2, B1, and B2 all interval operations ⊕ satisfy the following requirement:

`B1 ⊆ A1, B2 ⊆ A2 ↔ B1 ⊕ B2 ⊆ A1 ⊕ A2`

This is of interest because IRIS does not allow to represent split intervals like `[a, b] ∪ [c, d]` with `a < b < c < d`, without including all `x ∈ [b, c]`.

[63] Ramon E Moore et al. Introduction to interval analysis.

Figure 13/A: Example for interval arithmetic in IRIS. Although intervals only represent value domains, all common interval operations are implemented.
2.1 Roadmapping in IRIS

Figure 13/A presents an example for interval arithmetic. For A, the expression 
\(-1 + [1..4]\) evaluates to \([0..3]\) because for any possible number in the 
domain \([1,4]\), the following constraint holds: if it is to be reduced by 1, it now lies 
within the interval \([0,3]\).

Ternary Logic

Having value domains, tests that check if \(x\) “is” 3 can no longer definitely be 
answered by either true or false. Consider a case in which the narrowest 
inferable domain for \(x\) would be \([1,4]\). With \(3 \in [1,4]\) the answer to “is \(x\) equal to 
3” would be a “maybe”: there are possible values for \(x\) that can fulfill the equality, 
and there are possible values that can violate it.

To account for such cases, IRIS employs a ternary logic — similar to Kleene’s 
strong logic of indeterminancy [59] — using the constants true, false, and 
maybe. In the context of IRIS, maybe can be viewed as the boolean interval 
\([\text{false}..\text{true}]\). Table 14/A on the left side presents the definitions of the ternary 
operations “not”, “and”, and “or”.

Identifiers

IRIS differentiates two types of identifiers: i) those referencing a model element 
directly by its unique identifier, and ii) those reference a model element by 
its assigned name. No matter which type is chosen, only blocks, properties, 
and requirements can be referenced by the modeling user as they are the only 
model elements that are assigned an (observable) value: blocks through their 
availability (cf. Note 11/A), properties by their inferred value, and requirements 
by their condition.

To reference an element by its unique identifier, it has to be prefixed by the 
hash symbol. Therefore, \(3 + \#42\) references the model element with the 
unique identifier 42. Exploiting their uniqueness, references using the unique 
identifier can be used disregarding any hierarchy and solve the problem of name 
collisions.

Dates and Durations

Time plays a critical role for IRIS — it is treated as a first-class citizen. IRIS allows 
for literals like Feb2821 consisting of three letters representing the month, 
followed by four digits representing the year, to refer to a certain point in time, 
like “February, 2021”. Dates can be compared to other dates. For example, 
Aug2021 > Jan2021 yields true.

Using the functions years and months it is possible to define durations. (7) For 
each example, incrementing a date by a duration as in Jun2021 + months(6) yields 
the date after the duration: Dez2021.

The literal \(T\) allows expressions to refer to the current time in IRIS which can be 
set by the timeslider shown in Figure 10/A. Therefore, it can be used in 
expressions just like normal date literals. The expression \(T + \text{months}(2)\) results 
in a date that is two months later than the currently set time.

Moreover, every identifier (cf. Subsection 2.1.3) accepts a date as an argument 
which is implicitly set to \(T\) but allows for binding \(T\) to a different time locally. 
Showcased in Figure 14/A, writing \(A + \text{months}(2)\) is exactly the same as writing 
\(A(T) + \text{months}(2)\). However, writing \(A(T + \text{months}(2))\) binds \(T\) in \(A\) to \(T + \text{months}(2)\) evaluating \(A\) at a time that is two months from the currently set 
global time.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>!A</th>
<th>A&amp;B</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>f m</td>
<td>f m</td>
<td>f m</td>
<td>f m</td>
<td>m m m m</td>
<td></td>
</tr>
<tr>
<td>m f</td>
<td>m f</td>
<td>m f</td>
<td>m f</td>
<td>t m t m</td>
<td></td>
</tr>
<tr>
<td>t m</td>
<td>m t</td>
<td>m t</td>
<td>m t</td>
<td>m m m m</td>
<td></td>
</tr>
</tbody>
</table>

Table 14/A: Ternary logic, using t for true, f for false, and n for maybe. All other combinatorial values for A and B not listed work just like in boolean logic.

[59] Grzegorz Malinowski
Kleene logic and inference.

Note 14/A: Time Dependence.

<table>
<thead>
<tr>
<th>Intervals</th>
</tr>
</thead>
<tbody>
<tr>
<td>A: T Nov2021</td>
</tr>
<tr>
<td>B: A + months(2)</td>
</tr>
<tr>
<td>C: A(T + months(2))</td>
</tr>
<tr>
<td>D: B + Jan2938 false</td>
</tr>
<tr>
<td>E: (D Jan2948) true</td>
</tr>
</tbody>
</table>

Figure 14/A: Time dependence in IRIS.

(6) Note that IRIS bases the short forms on the german names. Therefore Mrz2030 would refer to march 2030.

(7) IRIS offers a wide range of operations and functions to deal with time. They exceed the scope of this explanation (cf. [15]).

Note 1.6/A: Time Dependence.

Subsection 2.1.3 on this page.
Using the argument allows binding $T$ to any date which allows querying for constant times as well. Consider $E$ which queries for the value of $D$ at the time \texttt{Jan2040} or in other words, querying for $B(\text{Jan2040}) > \text{Jan2030}$. Using the definition for $B$ and the definition for $A$ this results in the expression $\text{Jan2040} + \text{months}(2) > \text{Jan2030}$ that evaluates to true independently from the current global binding of $T$.

2.1.4 Solving

IRIS integrates a solver that infers values for all properties, requirements, availabilities, and more. It evaluates inheritances, hierarchies, the selection of solution spaces, and all expressions in three major steps [15]:

1. Lowering the model in a flat constraint system as a list of facts that all have to evaluate to true, abstracting all semantics stemming from the hierarchy.

2. Solving, by inferring closer and closer bounds for all elements.

3. Instantiating the simulated time and solving the remaining constraints.

While the details of the solving process are not of interest (cf. [15]), the following Example 15A should summarize the most critical aspects.

The model in Figure 15A on the right side contains three blocks: $A$, $B$, and $C$. Each of them has one property: $W$, that should represent the weight of the corresponding block. The weight of block $C$ is fixed to 3kg, whereas the weights of the blocks $B$ and $A$ are for the solver to infer.

The results are influenced by three constraint elements:

- $C1$ in $A$ enforces the result of $A.W$ to lie within 0 kg and 10 kg.
- $C2$ in $A$ restricts the weighed sum of $B.W$ and $C.W$ to the weight of $A$.
- $C1$ in $B$ restricts the weight of $B$ to be at least 0 kg.

Flattening the model, IRIS produces the constraint system shown below on the left. For readability reasons, unique identifiers are replaced by human-readable names and the replacement is ignored (cf. Note 11B):

\[
(A.W = [0 \text{ kg} .. 10 \text{ kg}]) \quad \Rightarrow A.W(T): [6 \text{ kg} .. 10 \text{ kg}] \\
(B.W + (2 \times C.W)) \leq A.W \quad \Rightarrow C.availability(T): \text{true} \\
(A.availability = \text{true}) \quad \Rightarrow A.W(T): [6 \text{ kg} .. 10 \text{ kg}] \\
(B.W > = 0 \text{ kg}) \quad \Rightarrow C.W(T): 3 \text{ kg} \\
(A.availability = \text{true}) \quad \Rightarrow B.availability(T): \text{true} \\
(C.W = 3 \text{ kg}) \quad \Rightarrow B.W(T): [0 \text{ kg} .. 4 \text{ kg}] \\
(C.availability = \text{true}) \quad \Rightarrow A.availability(T): \text{true} \\
\]

As this constraint system is not time dependent, the time instantiation does not affect results. The inferred bounds are shown in Figure 15A right of the $\Rightarrow$-icon.

Impossible Bounds

Depending on the model, constraints can contradict each other leading to the inference of impossible bounds. A simple example of this is shown in Figure 15B on the right side: While the expression of the property $X$ binds the result to true, the constraint $C1$ enforces the value to be false.

Impossible or — in other words — “empty” bounds, are visualized by the empty set: "∅" and defined in such a way that any operation with empty bounds yields empty bounds again. Therefore, $\emptyset$ always implies an error. It is not to
be mistaken for a legal value but quite the opposite: an element for which no value is possible. Dealing with empty bounds exceeds the scope of the proof of concept implementation in Chapter 4. In general, we do not consider models with any impossible bound as those errors distort the model behavior.

**Overwriting Behavior**

As noted in Subsection 2.1.1, every block has an implicitly defined replacement and an implicitly defined availability constraint. The former one is based on the existing solution alternatives, and the latter is defined through the children and requirements of the respective block (see Subsection 4.4.5).

However, it is possible to overwrite these implicitly defined constraints explicitly by using two special properties. The magic names of these unique properties are “Replacement”, and “Availability”. Figure 16/A on the left side showcases overwriting the blocks availability. Even though the only requirement of Overwrite is \( \text{false} \), its effect on the availability of the block is overwritten by the property with the name Availability. This can be identified by the background color of the block and by Test which references the availability of Overwrite explicitly. R1 has no longer any effect on the availability of the block Overwrite.

Using these overwrites has another effect. Because IRIS treats the overwriting model elements as ordinary properties with a “special” name, they can be referenced as any other property.

### 2.2 Formalization

Knowing how IRIS handles models, this section extends on the formalization from the introduction. Furthermore, it abstracts the expression language and its solver. In other words, this section provides a taxonomy for models, which is used in the model transformation approach in Chapter 3.

As a general overview, we consider a model to be a finite set of model elements from which we differentiate three different types:

1. **Structural model elements**

   Like the name suggests, structural model elements define the hierarchy of a model. In addition, a structural model element may contain any number of other model elements recursively and may provide them with additional semantics. It is capable of creating a tree-like hierarchy. The blocks known from IRIS (cf. Section 2.1) are structural model elements.

2. **Constraint model elements**

   These model elements contain an expression that is either taken as it is or transformed by a determining, finite process that depends on the specific type of constraint model element and may make use of the whole model (e.g, by making use of its location in the model structure).

   As an example for such a transformation, consider the properties from IRIS. They assign an expression \( E \) to a property named \( P \). If the block, that \( P \) is located in, is no implementation, this yields the constraint \( P = (E) \).

3. **Passive model elements**

   Passive model elements have no direct effect on the model behavior. They can serve as mere comments, or they can name certain relationships by functioning as an alias. However, they do not generate any constraint, so removing them does not affect the inferred value for any element, as long as the effects or potential aliases are resolved. One example for passive model elements in IRIS are notes.
All model elements are identified by a uniquely generated identifier that is used to reference them, although the model may prohibit referencing elements that do not resolve to any value (e.g., comments). Therefore, we distinguish model elements in another dimension by separating elements assigned a value by the solver from those that are not. We call elements — for which the solver infers values — “variables”. The set of all variables of a model \( A \) is denoted with \( V^E_A \).

### 2.2.1 Structure and Hierarchy

In contrast to \textit{IRIS}, the formal representation does allow for model elements to appear outside of a \textit{structural model element} like a block.\(^{(6)}\) Besides that, there are only two restrictions for additional semantics that may be provided by hierarchically structuring the model:

1. There exists a determining and terminating algorithm that can resolve all those semantics for a given model element.

2. All semantics added by the hierarchy have to be equivalently expressible without the hierarchy. Therefore, any model using structural model elements with additional semantics has to be transformable into a \textit{behavioral equivalent} model which does not make use of these additional semantics.

Both restrictions are necessary to create an obfuscated model that differs in its structure (e.g., hiding some hierarchy levels). For \textit{IRIS}, both restrictions are satisfied, and the respective algorithms are executed during the lowering step, that transforms the model into a flattened constraint system (cf. Subsection 2.1.4).

A model element distinguishes itself by its unique identifier, which may be but does not necessarily have to be in a human-readable format. For all examples, we allow human-readable aliases that are to be used instead of the unique identifiers. For those aliases, we use the following rules for readability:

1. The full name of an element \( a \) is defined by the joint name of all elements on the path from the model root to \( a \), using the dot as a separator. If \( a \) is inside of a structural element with full name \( \gamma \), the full name of \( a \) is \( \gamma.a \).

2. Model elements can always be referenced by their full name, no matter the scope they are referenced from.

3. A model element with full name \( \gamma.a \) can reference any other model element \( \gamma.\delta.b \) as \( \delta.b \). For any \( \gamma = \alpha.\beta \) this relative referencing holds, so that \( \gamma.a \) can reference any \( \alpha.\delta.c \) as \( \delta.c \).

Because this way of writing references is only for readability, we ignore any potential naming collisions and use unambiguous references only. The following Example 17/A illustrates the application of the formalization in the context of the children relationship that blocks may employ in \textit{IRIS}.

### Example 12/A

A model element with full name \( \gamma.a \) is defined by the joint name of all elements on the path from the model root to \( a \), using the dot as a separator. If \( a \) is inside of a structural element with full name \( \gamma \), the full name of \( a \) is \( \gamma.a \).

Model element: Uniquely identified element of a model.

Variable: Model element that receives a value by the solver.

Model: Set of model elements.

\( V^E_M \): All variables in \( V^E_M \).

Structural Model Element: Specific model element that defines the hierarchy.

IRIS allows for global “standalone” requirements, which exceed the scope of this thesis.

Behavioral Equivalence: All variables in common evaluate to the same value for all external factors.

\( \text{Subsection 2.1.4 on page 15.} \)

### Example 17/A

Formalizing the children relationship.

Example 17/A illustrates the application of the formalization in the context of the children relationship that blocks may employ in \textit{IRIS}.
2.2 Formalization

With this, one possible notation would be as follows:

\[
\text{Model} = \{
\text{Parent:} \{K: A.X + A.B.X, A: \{\ldots\}\}
\text{Block:} \{K: 2 * \text{Parent.K}, X: \text{Parent.A.B.X}\}
\}
\]

The next Example 18/A has a similar purpose than the previous one but explains the formalization of solution spaces instead.

To express interfaces and implementations, we can add a bidirectional linking between two blocks with special constraint model elements that have to occur inside of the structural model element “block”. One of those constraint model element is to link an implementation block to its interface block, and the other is to link the interface block to its implementation block. In combination with constraint model elements representing KPIs, they can control the constraint creation just like in IRIS. (cf. [15]).

Normalized Model

We use the term (normalized) constraint to refer to an expression written in the formal language from Subsection 2.2.2 that has to be fulfilled by the solver. In other words, a “fact” that is ultimately true. That is, a constraint reading \( a = 3 \) is not to be misunderstood by the mere expression \( a = 3 \) — with “a” being the unique identifier of a referenceable variable. While the constraint enforces the value of \( a \) to be 3, the latter one evaluates to \( a \) if \( a \) has the value 3, \( b \) maybe if \( a \) can have the value 3, or \( c \) false if \( a \) can not have the value 3.

Following this definition, the term normalized model refers to a finite set of normalized constraints that maps each of them to the model elements responsible for their creation. Furthermore, all implicit assumptions (like the availability of a block in IRIS, cf. Note 11/A) have to be expressed explicitly, by one or multiple normalized constraints representing their effects on the model behavior.

For a normalized model \( M \) we define \( E_M \) as the set of all origins mapped to by all constraint in \( M \). Consequently \( V^E_M \) refers to all variables in \( E_M \).

Based on this definition, every normalized model can be described by exactly one normalized constraint generated by joining all constraints using a logical “and”-operator. However, joining all constraints completely erases the use of an origin mapping and additional terms like “selected constraint” (cf. Chapter 3). Therefore, we define an invariant that is to be fulfilled automatically before and after any modification to the model: Every joined constraint in the form “\( C_1 \) & \( C_2 \)” is to be split into two separate constraints \( C_1 \) and \( C_2 \), maintaining the correct origins. As an example, \( a = 5 \) & \( b = 3 \) is split into two separate normalized constraints \( a = 5 \) and \( b = 3 \). We do not deal with this invariant explicitly but silently assume it is ensured with each modification to the normalized model. However, we allow weakening the invariant in the final stages of the obfuscation.

2.2.2 Formal Language

For most examples, we use the domain specific expression language from IRIS (cf. Subsection 2.1.3). However, the transformation descriptions in Chapter 3 target more abstract aspects of a formal language which is described in this subsection.
The formal expression language uses a human-readable text-based format that is expressed adequately by an AST according to the syntax rules described in the following segment. We assume that the text- and the AST-based representation are views on the same information. Therefore, any modification to the AST immediately reflects on the text-based presentation and the other way around. Furthermore, we allow exactly the variables of the model to be referenced.

Syntactic Elements

The language allows for numeric, boolean, and string constants. In addition, just like \( \text{I/R.sc/I.sc/S.sc} \), it allows for units and expresses all non-scalar values as an interval which describes the domain of the value in question.

Besides this, we describe all operations — arithmetic, relational, boolean, and others — as functions that have to fulfill the following requirements.\(^{(9)}\) The functions are described in more detail in the following segment "known functions":

- They have to terminate.
- They have to determine, all "functions" that must not determine are considered external factors (cf. Subsection 2.2.4).

Nonetheless, instead of writing \(+(3, 4)\) we use the known infix-notation for a lot of the ordinary arithmetic and boolean operations: \(3 + 4\). For the sake of simplicity, we do not consider any explicit loop constructs but represent them as a known function. Similarly, we write \(\text{if } P \text{ then } A \text{ else } B\) instead of \(\text{ifThenElse}(P, A, B)\).

Known and Unknown Functions

All functions directly supported by the solver are “known”. Labeling them as such allows for treating them differently by the obfuscation as we can exploit knowledge of their semantics, their contract, and their implementation. Besides that, we allow the user to define custom functions via special model elements — all of those user-defined functions are labeled as “not known” or “unknown”. Nonetheless, known and unknown functions have to terminate and determine for any possible input vector. Among the known functions are operations known from mathematics like \(+: (a, b) \mapsto a + b\) and \(*: (a, b) \mapsto a \cdot b\).

Known and Unknown References

We use the term “reference” for an unique identifier or an alias that refers to one specific model element (most likely a variable). The set of “known” references is modifiable at any time but defaults to all references that are valid inside the expressions of a model. When talking about the obfuscation, some transformation categories introduce new references while others modify the existing set of known references (e.g., the identifier obfuscation category).

Requirements

Two core requirements have to be fulfilled by any model:

1. Every implicit assumption that stems from the model and its structure has to be expressible explicitly using model elements. In other words, based on the constraints from the model, it should be possible to express the whole input/output behavior by a list of normalized constraints that do no longer have any implicit assumptions.

2. The behavior of a model determines for a fixed input vector of external factors. Furthermore, this behavior has to be resolvable by a solver in a finite amount of time. The characteristics of this solver are discussed in Subsection 2.2.3.
2.2 Formalization

2.2.3 Formal Solver

The formal solver has to infer values for model elements by evaluating the expressions in context of the model and all external factors. For every model element where there is not one sole value that can be inferred, the solver infers the closest valid domain that the result has to lie within. Besides that, the behavior for contradicting constraints is undefined.

We always assume that the solving process terminates (albeit this can be achieved trivially by stopping it after, e.g., a fixed amount of time) and that it works correctly.

Furthermore, we do assume that the solver is keeping track of the possible domains of all variables allowing us to inquire the closest possible domain the value of any variable has to lie in to not invalidate the constraints at any time.

Based on this, the following segment clarifies the definition of the term behavioral equivalence.

Formal Behavioral Equivalence

As motivated in the introduction in Subsection 1.2.2, we require a mechanism to compare the input/output behavior of two or more models — which allows making statements about a model and its one-way transformed counterpart.

Therefore, we compare all variables that both models have in common. To allow for a more flexible obfuscation, we allow for a bijective function \( m \) that maps all unique identifiers from the identifier domain of one model to the identifier domain of the other model. We write the variables both models have in common after applying the bijective mapping function \( m \) as \( V^E_A \cap m \cap V^E_B \).

Furthermore, we allow all external factors (like the global time in Iris) to be restricted by a finite domain. The definition of behavioral equivalence for a time \( t \) as the only external factor reads as follows:

\[
\forall t \in [t_s, t_e] \forall e \in V^E_A \cap m \cap V^E_B : e_A(t) = e_B(t).
\]

We use \( e_A \) and \( e_B \) to identify variables that have the same mapped identifier but stem from two different models. Therefore, \( e_A \) refers to the variable \( e \) from the model \( A \) and \( e_B \) refers to the variable \( e \) from the model \( B \). Writing \( e_A(t) \) and \( e_B(t) \) we refer to the closest value the formal solver was able to infer for those variables at the given point in time \( t \).

This definition does exclude side effects that exceed the model scope (like the creation of files), as we do not consider them as part of the input/output behavior. This is consistent with the behavior definition from Collberg, Thomborson, and Low if we reduce the “observable” part of a model to be the model itself [25].

Tolerable Behavioral Equivalence

Tolerable behavioral equivalence is a formal extension of the behavioral equivalence definition. It exchanges the equal comparison of \( e_A(t) = e_B(t) \) by an arbitrary function “\( \text{comp}_e(x, y) \)” which may be defined for each variable \( e \in V^E_A \) and defaults to \( x = y \):

\[
\forall t \in [t_s, t_e] \forall e \in V^E_A \cap m \cap V^E_B : \text{comp}_e(e_A(t), e_B(t)).
\]

With all other definitions unchanged, \( \text{comp}(x, y) \) is true if and only if the difference of \( x \) and \( y \) is “tolerable” — an abstract concept inferred by external rules, e.g., tolerances of the production process. This could be a comparison with an absolute value if \( x \) and \( y \) are both numbers. Consider this example with the fixed tolerance-parameter \( \alpha \): \( \text{comp}(x, y) = |x - y| < \alpha \).
2.2.4 External Factors

We consider everything defined outside of the model and its expressions to be an external factor. In the context of Iris, there is only one external factor that we do consider in this thesis: time. Nevertheless, different modeling languages may grant access to other external factors like a random number source or even outsource parts of the calculations to another standalone system.\(^{(10)}\) Even though we do not discuss them explicitly, they can be dealt with similar to time by considering them as unknown factors with a fixed domain and applying transformations like Transformation Category 10. However, we do not deal with external factors that rely on a calling order like a generator that deterministically outputs another number after each access.

External factors are allowed if a determining and terminating algorithm can identify their influence precisely. For every expression and model element, the influence of external factors must be identified clearly. Furthermore, the solver must be able to evaluate these external factors. Discussions on how the solver should evaluate those external factors — for example, by sending specific requests to other servers — exceed the scope of this thesis.

2.3 Model Transformation

Primarily due to the trend in model-driven architectures (MDA), there are a lot of different approaches to model transformation [74]. They can be grouped in various ways, for example, if they deal with PIMs or PSM. We use the two major categories proposed by Czarnecki et al.: model-to-model and model-to-code approaches [29]. However, because our approach of a one-way model transformation mainly falls into the former category, we restrict the scope of this brief overview to model-to-model approaches. Other approaches are thoroughly analyzed and compared by Syriani [79, 1.1.3].

Model Transformation Languages

There exists a wide variety of languages that work as abstractions on the model transformation process. Some of them, like VIATRA2 [82], focus on graph-based models. Others, like MOMENT, use a mathematical founded approach [14, 22]. Independently from their focus and theoretical foundation, some use a graphical syntax [84] while others, like the well-known QVT standard or ATL, use a textual representation [78, 53].\(^{(11)}\)

Our Approach

When specifying the transformations in Chapter 3 we do not make use of any of the model transformation languages mentioned but follow a hybrid approach using pseudocode and abstract descriptions (cf. [29]). There are three main reasons for that.

First of all, while the overall approach is a one-way model-to-model transformation, we allow for behavior to be embedded inside the model elements (as stated in the formalization in Section 2.2). This embedded behavior results in a process that starts by describing the model behavior in a constraint system. Then it obfuscates that constraint system and finishes by transforming it back into a model that represents that constraint system again, further obfuscating the structure of that model (cf. Section 3.1).

Second, the abstract notation we use is not bound to any concrete syntax rules and limitations of using a predefined language. We operate using only formalized terms allowing for great flexibility when implementing the transformations MDA: Model-Driven Architectures

MDA: Model-Driven Architectures

ATL: A QVT-like Transformation Language

Frédéric Jouault et al.

ATL: A QVT-like Transformation Language

Dániel Varró et al.

Model Transformation Language

A Multi-Paradigm Foundation for Model Transformation

Henshin: A Usability-Focused Framework for EMF Model Transformation

Maude: Specification and programming in rewriting logic.

Robert Wagner

Model Transformations with Fujaba.

Daniel Strüder et al.

Henshin: A Usability-Focused Framework for EMF Model Transformation Development.

Frédéric Jouault et al.

ATL: A QVT-like Transformation Language.

(10) In fact Iris has more external factors, like the runtime prediction, which exceeds the scope of this thesis.


\% Chapter 3 on page 27.

\% Section 2.2 on page 16.

\[21\]
presented. For example, a concrete implementation could very well use ATL or another language to describe the transformations. However, our proof of concept implementation mostly uses direct symbolic transformations to stay consistent with the existing IRIS implementation.

Third, recent research has shown that a lot of businesses do still use general-purpose languages “probably, because there is no convincing evidence that dedicated [model transformation] languages are substantially better” [17].

2.4 Related Work

While there exists a lot of research on related problems, we are not aware of any research dealing with the very same problem as described in Section 1.2. We have loosely divided the related work into four groups. Those whose focus lies on i) model transformation, ii) anonymization of data, iii) obfuscation of code, and iv) those who deal with the formalization and verification of transformation approaches in general.

However, while we can not prove that there is no previous research on the problem statement as stated, we can explain our research strategy for subsequent readers to understand the steps taken.

Research Strategy

We started with several extensive literature surveys and systematic literature reviews already conducted in the area of obfuscation [8, 52, 1, 89]. For example, Hosseinzadeh et al. [52] covered 357 articles selected from thousands of papers but excluded, amongst others, work on protecting IP.

However, even following up with papers mentioned in those sources and the related works they have, most of them focus on the obfuscation of complete programs or the removal of personal information from large datasets. Furthermore, many obfuscations and anonymization techniques do not require the output to be readable or in the same format. They either only retain metadata required for subsequent analysis or use an assembler or binary format for representation [52].

Those operating differently almost exclusively use a single trusted or a set of multiple parties that receive extracted parts of a program or model to answer requests in an oracle fashion. However, those approaches no longer yield a standalone program. Instead, they require online work to communicate with the other parties.

Thoroughly analyzing over 100 surveys, papers, and technical reports, we list a subset of those closely related to our problem of one-way model transformation in the remainder of this section as a basis for further research. Additional sources are cited in Chapter 3.

2.4.1 Model Transformation

Research of this field deals with the uni- and bidirectional transformation of one or several models into one or several models of an at least comparable abstraction level [30].
**Business Process Model Obfuscation**

Goettelmann et al. propose an approach on obfuscating Business Process Models (BPMs) at design time. In their paper, they present a block based fragmentation of BPMs using obfuscation constraints [48]. Their goal resembles the one of this thesis because they want to avoid that a single cloud provider used by the company can understand critical fragments of a BPMs.

Nevertheless, their way of achieving this goal differs as they do not obfuscate any contents or remove any information from the whole model. Instead, they split a single business process model into several fragments that can be distributed to different cloud providers which have to communicate with each other during the execution. Therefore, any fragment sent to a cloud service provider or another company requires communication with other model parts to be executed and to behave equivalently. Additionally, there is no comparable “selection” mechanism as the approach aims to be (apart from a set of QoS/security requirements) fully automatic.

Figure 23A showcases the structure of such a fragmentation, “Send” and “Get” represent the communication with other cloud providers. The name of actions like “GLA” have been shortened as their semantic meaning exceeds the scope of this summary (cf. [48])

**Obfuscating CAD Models**

The goal of Gupta et al. is to protect CAD models against counterfeiting to protect intellectual property (IP, [51]). However, they follow a quite counterintuitive approach using their “ObfusCADe” protection strategy to introduce additional features into the CAD models. Those additional features are not to dilute other features in the model but rather to “sabotage” reproductions based on this model. They exemplify this by inserting spheres in a CAD-model. Those spheres heavily decrease the quality of parts produced by printing those models.

**Obfuscating Conceptual Models**

The author Fill has the same goal of supporting IP-preserving collaboration in the context of models [38]. Although he focuses on conceptual models [54], a lot of transformations he describes are applicable in the context of our problem statement. However, he remains on an abstract level presenting abstract examples of what could be done but not when to apply or how to perform such transformations automatically, or what problems their application implies.

The author groups transformations into four different categories. Representation transformations are generally concerned with changing the layout, the scrambling of identifiers, and the removal of elements. Structural transformations are used to aggregate elements or completely remove “alternatives”. They resemble the removal of unselected information that we propose but do not care about changes in the behavior. Data transformations resemble those found in privacy-preserving data mining (PPDM, [46]). For example, they can remove all names of company employees or exchange all Internet Protocol addresses by random other ones. Semantic obfuscation transformations use
2.4 Related Work

Data Anonymization is closely related to the area of privacy-preserving data mining \((PPDM, \ [46])\). The research presented in this group focuses on the anonymization of either a single datum or data in general, but not control logic. Although we do not focus on the replacement of data (e.g., removing all names of company employees), research in this group may be applied additionally.

AnonymousXL

In “Anonymizing Spreadsheet Data and Metadata with AnonymousXL” the authors Van Veen et al. briefly present their work on AnonymousXL \([81]\) — an addition to their analysis tool PERFECTXL \([72]\). While their approach focuses on EXCEL-spreadsheets, it deals with similar problems: the anonymization of confidential data without impeding a subsequent analysis to an intolerable extent. Yet, they ignore formulas, embedded constants, and even some numbers for the sake of analysis results. Therefore, their process is limited to i) metadata which is set to a constant value, ii) sole magic numbers that are randomly shifted in a given tolerance interval, iii) dates which are replaced randomly, and iv) textual data that is replaced by random textual values keeping their uniqueness.

Single Datum Obfuscation

In their paper “Data obfuscation: Anonymity and desensitization of usable data sets”, the researchers Bakken et al. create a foundation for the obfuscation of data and categorize techniques by three properties \([7]\): i) reversibility, ii) specification, and iii) shift. Additionally, they feature some example primitives that can be used for a simple obfuscation and assess their usefulness in an experiment. An overview of the major categories is presented in Figure 24A on the left side. While they focus on the obfuscation of a single datum (e.g., a numerical value), they discuss the obfuscation of data structures and more complex data types. However, even this discussion is restricted to combined data types and does not consider control structures and models. We extend on their concept of reversibility in Subsection 3.2.2 and use similar techniques in Transformation Category 1 and Transformation Category 2.
Anonymizing Biomedical Data
Similar to AnonymousXL, “ARX” is a tool used to remove private information [68]. However, the focus of ARX lies on removing this information from medical information. The authors Prasser et al. follow a more formal approach by using the privacy criteria of k-anonymity, ℓ-diversity, t-closeness, and δ-preservation. ARX is a fully implemented program [13] offering a wide range of transformations that are selected based on the application scenario. Similar to ARX, there is a wide variety of tools that focus on anonymization in various domains. As an example, Tarzan is a tool that anonymizes senders and recipients using a network address translater (NAT) when communicating over the Internet [41].

2.4.3 Code related obfuscations
Code-related obfuscations do not deal with models or static data in tables (e.g., inside an Excel spreadsheet) but with code. The acquired methods make use of access restriction, control-flow, and information obfuscation [8].

Mondo
In “Secure views for collaborative modeling” the authors Debreceni et al. describe the insufficient administrative capabilities of source code repositories and introduce their MONDO-framework for rule-based access control policies [34, 33, 32]. They essentially provide views of a source code repository that reveal only parts of the whole underlying model so that several crucial or secret aspects of a code are not revealed to industry partners.

However, their approach differs on several crucial points. First of all, they assume the existence of a so-called “gold model” that contains all of the information. Additionally, they describe the “front models” as mere secure views on this gold model and not as standalone and independent versions of it. That contradicts the C/1 constraint for the Huskied Model, requiring it to be self-contained (cf. Subsection 1.2.3). Furthermore, their concept of a “lens” (cf. [12]) as a bidirectional model transformation is not guided by a selection of elements but by a collection of access rules. Those access rules only affect the Get and PutBack operations that represent the interface between the gold model and its fronts.

Compiler Optimizations
While compilers usually do not obfuscate the code they compile, they use several transformations that aid our one-way model transformation purpose [26]. Especially optimizations that remove unreachable or dead code and that already pre-evaluate parts of a program. They hide calculations and remove origin information by inlining constant values. For example, we adapt the well-known compiler optimizations of constant folding and constant propagation and introduce them as T.4 Arithmetic Simplification and T.5 Inline Definitions in Chapter 3.

Hiding Design Intent
The authors Sosonkin, Naumovich, and Memon focus on a subset of program obfuscations by hiding the design intents embedded in object-oriented applications [77]. They present three transformations called i) class coalescing, ii) class splitting, and iii) type hiding. Furthermore, they provide a design obfuscator for Java using all three obfuscation techniques.
Even though the formalized models we use do not have the concept of classes, similar transformations can be applied to the structure. For example, by splitting a block in two or merging two blocks (cf. Section 5.3).

2.4.4 Formalization and Verification

In this final segment, we give a brief overview of research on methods to measure various properties of an obfuscation process. They extend those found in the large verification survey conducted by Ab. Rahim et al. [1].

The authors Banescu et al. present a framework to analyze the obfuscation resilience of software against automated attacks [9]. Next to the construction of a formal model, they perform a case study in which they use KLEE [20] — a symbolic execution tool for automated test generation — to retrieve valid licenses by generating test cases based on the LLVM\(^\text{14}\) bitcode of a license checker program.

While Viticchié et al. only test one obfuscation technique (called VarMerge), they present an experiment design involving real persons as the “attackers” measuring the correctness and required time for the attack [83].

Regarding formalization, Asztalos, Lengyel, and Levendovszky present an assertion description language for model transformations based on graph rewriting, that allows for easy verification [6]. As another example, the functional mockup interface (FMI) is a standardized format for the exchange of simulation models [5]. FMI is common in the automobile industry and uses XML definition files alongside functions written in C that are responsible for the Co-Simulation.

\(\text{LLVM}^\text{14}\)

\(\text{FMI}^\text{14}\)
3 One-Way Transformation

As the formal foundation, this chapter provides an overview of the proposed program as well as a catalog of all the presented transformations alongside their analysis. The main focus lies on the model transformation aspect. Therefore, user interaction is neglected entirely and deferred to the proof of concept implementation later. For now, we assume that all preconditions of the first transformation phase are met and that all postconditions of the last transformation phase are of use.

3.1 Overview

To solve the problem as formulated in the problem statement, we propose a program which is split up into four distinct and sequentially executed phases, that are as follows:

Phase 1: Normalize the input model and selection.
Phase 2: Prepare and populate supporting data structures.
Phase 3: Obfuscate by applying transformations until exhaustion.
Phase 4: Verify the produced result.

The central part of this chapter focuses on the second and the third phase, proposing and analyzing various transformations to achieve the desired obfuscations. Refer to Chapter 4 for a closer look at one specific implementation of the other two phases, 1 and 4.

Figure 27/A below acts as a brief overview of all four phases. Some phases, like the third, are split further into subphases which are explained in more detail in the following subsections.
3.1 Overview

3.1.1 First Phase – Normalization

The first phase is closely related to the implementation domain as it must know the implicit assumptions and rules of the model language for which the transformation process is to be implemented (cf. Figure 28/A on the left side). Therefore, this phase is to be initialized by the model, the selection of model elements, and as much additional information from the model language as is required to meet the preconditions of the next phase.

Important Aspects

The required normalization steps depend on the structure, semantics, and the concrete definitions of the model. While the latter one is already described in Subsection 2.2.3, the structure and semantic rules may be expressed in any way. The following points serve as an overview of important normalization steps:

- If the model contains a hierarchy of model elements (cf. Subsection 2.2.1), this hierarchy must be removed for the following phases by flattening the model. Nevertheless, it is possible to record the original structure if required using the “tracing model” explained below.

- If the selection is governed by expansion rules — for example, the selection of a structural model element always implies the selection of all of its direct children — the selection has to be fully expanded with respect to the implicit assumptions which are explicitly expressed in the normalized model. Therefore, there have to be rules which decide, for example, whether any implicit assumption about a selected model element is selected as well.

- The input/output behavior of the model has to be “normalized” by transforming it into a set of normalized constraints that are mapped to the model elements they originate from. We call this set of mappings a normalized model. The normalized model has to perfectly represent the behavior of the model. It is not allowed to contain any implicit assumptions (cf. Section 2.2). The normalized model enables the identification of selected and unselected constraints. We consider a constraint of the normalized model to be “selected”, if at least one of its origin model elements is selected. Otherwise, we consider a constraint to be unselected.

Example 28/A: Removing implicit assumptions.

Consider the model: \( M = \{ a : 42 \} \), accompanied by the selection \( S = \{ a \} \). For this example, we consider the model to be not normalized in its current form. More precisely, we assume that the arbitrary input semantics enforce another model element \( \rho \) for each element, which represents its priority and is implicitly set to 0 if not given explicitly.

The normalized model \( M' \) — which is produced by this phase — must therefore contain this element: \( M' = \{ a : 42, a.\rho : 0 \} \). It is up to the selection and the source semantics if this element is to be appended to the selection. For this example, we consider the element to be excluded from the explicit selection \( S = \{ a \} \). If we had decided otherwise, the selection would cover all of the model elements in the flattened model, which would allow omitting the obfuscation process.

Note that this example does not use a constraint-based model-representation for clarity reasons. Example 29/A illustrates the process of flattening a model using a constraints-based representation.
Implicit Assumptions

There is an essential point to be made about implicit assumptions. They can make the process more difficult if they can not be expressed directly in the modeling environment. For example, consider a modeling language that requires conditionals written by the user to have always a “then”, and an “else”-case in the form of \( \text{if } X \text{ then } A \text{ else } B \). If the obfuscation requires removing the “else”-case from a statement — to remove the embedded data — we have to construct a dummy “else” which neither invalidates any other rule of the modeling language nor reveals any information about the removed case. Depending on the additional requirements this “else”-case has to fulfill (e.g., being a valid IP address that is currently unused in the model), this can be arbitrarily complex.

To avoid such problems in the formal approach, we do assume that the input language does only have implicit assumptions which are directly expressible in an explicit form — just like the “priority”-property in Example 2B/A. Furthermore, we do expect that cases like a mandatory “else”-case can be handled.

Flattening and Constraint Generation

We allow the input model to have a hierarchy that may even be of interest for implicit assumptions. For example, an internal model element that counts the number of children a parent element has. Because the obfuscation is mostly hierarchy agnostic, we have to express this internal element explicitly in the flattened normalized model.

The following Example 29/A presents a simple model and its flattening.

We consider a hierarchical source model that uses a model element named “block” to group various model elements (just like Ir18). One block may have an arbitrary amount of model elements assigned to it, including other blocks. Blocks inside a block are said to be “children” of the “parent” block, and every block may only have one parent at most. Those blocks serve as the equivalent of a namespace in modern programming languages, so we use the common notation using a dot-separated syntax to access elements within blocks (as described in Subsection 2.2.1). Figure 29/A on the right side showcases a simple hierarchical model following this concept. There is a parent block (named “Parent”) and two children-blocks “C/one.tf” and “C/two.tf”. Furthermore, there is another block named simply “Block” that references other model elements (e.g., Parent.C2.x).

Flattening. We flatten this model by resolving all identifiers to their “absolute name” using the relative notation allowed by the formal language (as seen by the enumerated constraints on the right side). If we take a look at \( x \) in “C1”, the accompanying constraint becomes \( \text{Parent.C1.x} = 3 \) as “C1” itself is a child of the “Parent” block.

Mapping. The origins of a constraint depend on the flattening and assignment semantics. For the example model in Figure 29/A each constraint has exactly one origin: 1 \( \mapsto \) Parent.a, 2 \( \mapsto \) Parent.b, 3 \( \mapsto \) Parent.C1.x, 4 \( \mapsto \) Parent.C2.x, and 5 \( \mapsto \) Block.a. Therefore, the normalized model is:

\[
M = \{ \text{Parent.a} = 2 + \text{Parent.C1.x} \mapsto \{\text{Parent.a}\}, \\
\text{Parent.b} = 3kg \mapsto \{\text{Parent.b}\}, \\
\text{Parent.C1.x} = 3 \mapsto \{\text{Parent.C1.x}\}, \\
\text{Parent.C2.x} = 5 \mapsto \{\text{Parent.C2.x}\}, \\
\text{Block.a} = 3 \mapsto \text{Parent.C2.x} \mapsto \{\text{Block.a}\} \}
\]
Tracing

The goal of tracing is to enrich the i) transformation categories of the catalog in Table 38/A, and ii) the verification phase with necessary information about the original model and all modifications. On an abstract level, it is a collection of graphs that represent all modifications to the original model elements. However, in practice, the tracer is a tool to aid the subsequent phases and therefore may have any form that suffices this need. In this thesis, the tracing model is used to store/record four things:

- **Tracer/1**: The original structure of the source model.
  For the sake of simplicity, we include the whole original model to retain its structural information. We will omit it when denoting the tracing model.

- **Tracer/2**: A mapping of identifiers from the source to the target model.
  To denote the mapping function of unique identifiers, we use a simple list of relations in the form “old \(\mapsto\) new”. Furthermore, we assume the mapping is bijective and can be queried in both directions.

- **Tracer/3**: Definitions and the configuration of the modeling environment.
  While this may include information like global definitions, implicit assumptions, and more, we do not include them in the denotation of the tracing model but reference them briefly whenever of importance.

- **Tracer/4**: Obfuscation operations applied to the model.
  Those “operations” refer to every modification that happens to the model during the obfuscation. While those operations can be written in a verbose but clear form (e.g., “Changed the formula of model element ‘X’ from \(X = 2 + 3\) to \(X = 2 + 4\)”) we reduce them to a more readable format most of the time. Therefore, we write \(X = 2 + 3 \mapsto X = 2 + 4\). Additionally, we write a sequence of operations in a chain: \(a + b \mapsto b + b \mapsto 2 \times b\). In most cases we just write, that an expression before and after a modification is to be stored, denoting the automated addition of respective elements.

The following Example 30/A showcases the construction of a simple tracing model for the process of an identity preserving reference obfuscation transformation category.

---

**Example 30/A**: Simple tracing model.

For this example, we ignore the previously presented program from Subsection 3.3.1 and consider a case in which all unique identifiers of a model have to be obfuscated. As input, we expect a set of constraints that may contain references to model elements in the form of strings. The output has to be the same set of constraints, only differing in the strings representing those references. They are to be replaced by random strings while preserving the identity of the element referenced originally.

So we expect a transformation as follows (based on Example 29/A):

\[
\begin{align*}
\text{Parent} \cdot a &= 2 + \text{Parent} \cdot C1 \cdot x & P1 &= 2 + P2 \\
\text{Parent} \cdot C1 \cdot x &= 3 & P2 &= 3 \\
\text{Block} \cdot a &= 3 \times \text{Parent} \cdot C1 \cdot x & P3 &= 3 \times P2
\end{align*}
\]

The accompanying tracing model has the following three mappings:

\[\text{[Parent} \cdot a \mapsto \text{P1}, \text{Parent} \cdot C1 \cdot x \mapsto \text{P2}, \text{Block} \cdot a \mapsto \text{P3}].\]

The order and abstraction of operations are not fixed and depend heavily on the concrete algorithm used and may vary in order and detail. The following is just an example of an algorithm handling the encountered statements from top
3.1.2 Second Phase – Preparation

The preparation phase is responsible for the initialization of all data structures necessary for the transformation categories of the third phase. Therefore, the concrete course of action that is to be taken in this phase heavily depends on the categories selected for the following phase.

From an abstract perspective, this phase expects a quadruple \((N, S, T, M_\tau)\) produced from the first phase and produces a quintuple by decorating the input quadruple with an additional tuple \(D\) — that may be named for readability — that contains the desired data structures. What is roughly sketched in Figure 31/A on the right side is detailed further in the following paragraphs.

**Preconditions**

The 2\(^{nd}\) phase expects a tuple consisting of four components: \((N, S, T, M_\tau)\). The components are i) a normalized model \(N\), ii) the selection of model elements \(S\), iii) a time interval \(T\), and iv) a tracing model \(M_\tau\). The time interval might be unbounded if time is of no importance (cf. Subsection 2.2.4), and the tracing model may be empty if the previous phase did not need to record any of its steps. Furthermore, the following constraints have to hold for \((N, S, T, M_\tau)\):

- **Pre/2.1** \(S\) is well formed (\(\forall e \in S \implies e \in E_N\)).
- **Pre/2.2** \(S\) is neither empty nor the complete model (\(S \neq \emptyset, S \neq E_N\)).\(^{(i)}\)
- **Pre/2.3** \(T = [a, b]\) is well formed (\(a \leq b\)) or unbounded (\(T = (-\infty, \infty)\)).
- **Pre/2.4** \(M_\tau\) contains traces for all modifications from the previous phase.

If any of the constraints is violated or any of the components is malformed (meaning “not well formed”), the behavior of this phase and all subsequent phases is considered to be “undefined”.

Note that even if we only refer to “time” as the only input, we use the term as a representation of all external factors that the modeling languages provide access to. We do this because we rarely have to deal with individual external factors differently — in fact, the only case would be T.10 Tailor Expressions which may employ individual tailoring algorithms for each external factor. Therefore we address them collectively as “time” allowing for simpler examples and an easier resemblance with the features of TRIS.

The following Example 31/A presents one possible input for the preparation phase. Additionally, it shows that the input validates all four preconditions.

Consider the model: \(M = \{a: 42, b: 21\}\), with the selection \(S = \{a\}\), and the unbounded time \(T = (-\infty, \infty)\). Furthermore, assume that the normalization of the first phase results in the normalized model \(N = \{a = 42 \rightarrow a, b = 21 \rightarrow b\}\). The input for the second phase is:

\[
\left(\begin{array}{c}
\{a = 42 \rightarrow a, b = 21 \rightarrow b\}, \{a\}, (-\infty, \infty), [ ]
\end{array}\right)
\]

\(N\) \(S\) \(T\) \(M_\tau\)
This satisfies all constraints Pre/2.1 – Pre/2.4 almost trivially:

1. \( \forall e : e \in S \implies e \in E_N \)  \( \text{Pre/2.1} \)

   With \( S = \{a\} \) and \( E_N = \{a, b\} \) it follows that \( a \in \{a, b\} \).

2. \( S \neq \emptyset \land S \notin E_M \)  \( \text{Pre/2.2} \)

   It holds that \( S = \{a\} \neq \emptyset \) and that \( S = \{a\} \neq \{a, b\} = E_M \).

3. \( (T = [a, b] \land a \leq b) \lor (T = (\infty, \infty)) \)  \( \text{Pre/2.3} \)

   With \( T = (\infty, \infty) \) the time is exactly the second possible construct.

4. \( M_\tau \) contains traces for all modifications.

   Due the assumption, the first phase did nothing but producing the trivial normalized model. We assume \( M_\tau \) to be trivially empty.

### Postconditions

Initialized with the input \((M, S, T, M_\tau)\), phase 2 is to emit a quintuple of the form: \((M, S, T, M_\tau, D)\), with \( D \) representing a tuple of required data structures. As this phase does not change any input data, it may be viewed as mere decoration of the normalized model.

The set of concrete postconditions depends on the data required by the transformation categories selected. If none needs to be added, there are none besides the existing preconditions (as nothing has changed). The following Example 32/A showcases one possible data structure to populate.

For this example, we assume an AST-based representation of the constraints, so we have some kind of tree structure that represents all tokens of importance. Furthermore, we assume the following input:

\[
\left( \{ a = 42 \mapsto a, \ b = 21 \mapsto b \}, \{b\}, (\infty, \infty), [ ] \right)
\]

Traversing through all elements \( e \in E_N \) of the model, this phase may create a function \( \text{id}_N : E_N \to \mathcal{P}(\text{pos}(E_M)) \) mapping each unique identifier of a model element to all of its occurrences (which are referred to by pos) in the normalized constraints. We use \( \mathcal{P}(X) \) to denote the powerset of a set \( X \). For this example, artificial positions are presented in Figure 32/A on the left side by small numbers next to each node. They coincidentally represent the visiting order of a breadth-first-search in the AST. Therefore, the example input yields the function:

\[
\text{id}_N : E_N \to \mathcal{P}(\text{pos}(E_M)), \quad a \mapsto \{\text{pos}(b, 1)\}, \quad b \mapsto \emptyset
\]

With \( a \not\in S = \{b\} \) this allows the subsequent phase to perform name obfuscation on \( a \) by replacing the identifier at all occurrences in an identity-preserving way.

Note that all transformation categories that modify, create, or remove references have to maintain the integrity of this data structure.

### 3.1.3 Third Phase – Obfuscation

This phase uses the quintuple \((N, S, T, M_\tau, D)\) of the phase 2 and executes a predefined set of model transformations according to their execution rules. Therefore, the preconditions match the postconditions of the previous phase and are not repeated here. The input and output of this phase are sketched in Figure 33/A on the facing page.
In more detail, this phase is split up into four different subphases. These subphases are executed sequentially according to the execution rules described in Section 3.5 and perform the following operations:

1. **Outer** obfuscation phase.
   Deals with unselected **normalized constraints**.

2. **Inner** obfuscation phase.
   Handles the effects of unselected normalized constraints on unselected constraints.

3. **Structure** obfuscation phase.
   For hierarchical models, it focuses on the model-dependent structural properties. In general, this phase builds the “new” output model based on the normalized constraints.

4. **Cleanup** phase.
   Removes redundant **model elements** and refines the models appearance. Furthermore, it may be used to recover selected elements that have no influence on the models behavior.

The examples and descriptions for the obfuscation phase vary in their abstractness to cover various model representations and possible additional transformations that improve or worsen the readability of the result. However, we try to avoid leapin ahead to any characteristics of the implementation in Chapter 4. The abstract expectations for the individual subphases are summarized further in Note 33/A.

The ultimate goal of the obfuscation phase is to generate a new model $M'$ from a **normalized model** $N$. To achieve this, the first two subphases outer and inner only modify the constraints in the normalized model. During their execution, $M'$ does not exist so their modifications only change the normalized constraint-representation of the original input model $M$. The creation of $M'$ is deferred to the **structure** subphase that uses the modified constraints to generate a new model that behaves just like $N$. Finally, the **cleanup** subphase only operates on the model $M'$ produced by the **structure** phase and applies potential post processing operations.

**Postconditions**
Phase 3 initialized with the input quintuple $(N, S, T, M, D)$ emits a transformed quintuple $(M', S, T, M', D')$ through a set of predefined model **transformation categories**. Note that while the input is expected to be a normalized model, the output $M'$ is the obfuscated model $M' = B_{A,S}$ for the original source model $A$. Therefore, this phase consumes a set of normalized constraints and produces a model on the same abstraction level as the original input model. The following postconditions have to hold for each well formed input:

- **Post/3.1** $B_{A,S}$ contains no reference to any $e \in E_A$. \hfill (C/1)
- **Post/3.2** $B_{A,S}$ contains all elements from $S$. \hfill (C/2)
- **Post/3.3** $B_{A,S}$ and $A$ are **behavioral equivalence**. \hfill (C/4)

However, these conditions exclude $C/3$ which restrict the amount of information about $A$ that can be inferred from $B_{A,S}$. This is due to its limited verifiability of this constraint which will be explained further in Section 3.4.
3.2 Classification

This section introduces how the individual transformation categories and the overall transformation approach are analyzed. We use the term transformation category instead of transformation to recognize that one transformation category — which fulfills a single purpose like flattening the model — may be achieved by a set of multiple transformations working together.

There are many aspects by which transformations and whole model transformation approaches can be analyzed. In “Classification of model transformation approaches”, the authors Czarnecki and Helsen present a taxonomy to classify model transformation approaches, by providing hierarchically structured feature diagrams that focus on the specification of the transformations themselves. Another (closely related) taxonomy framework is proposed by Mens and Van Gorp and follows a multi-dimensional approach whilst proposing additional "success criteria" for a transformation language. This thesis makes use of both frameworks to classify the overall transformation approach in Subsection 3.2.1 and the scheduling rules in Section 3.5.
3. One-Way Transformation

3.2.1 Overall Allocation

From an abstract perspective, all transformations considered in this chapter are to be execute in the obfuscation phase and are applied according to the execution rules described in Section 3.5. They form the basis of the model-to-model approach to solve the problem statement. We consider two kinds of transformations: (i) those that change the normalized constraints, and (ii) those that create and modify the structure of the output model.

All transformation categories are presented in the following fashion: First, there is an introduction motivating the need for the category (see Note 37/A) and explaining the required prerequisites alongside the relations with other categories. Afterwards, there is a description of the algorithm for each individual category. If internals are of interest, the description is given as pseudocode alongside a contract defining what the category requires and what it achieves. On the other hand, if further details do not serve any significant benefit or if there is already a lot of research regarding possible implementations, we do only present a textual description. Based on the description we finally discuss the properties of the category (cf. Subsection 3.2.2) and its influence on the four constraints of the problem statement.

The authors Amrani et al. list common intents of transformations [4]. Using their taxonomy, the main transformation intent is the manipulation of the source model. However, the authors do not consider transformations used with the intent to anonymize or obfuscate specific data on the same abstraction level — which is our primary intent.

3.2.2 Transformation Properties

This thesis recognizes a set of properties to be analyzed for each transformation individually. These properties are based on the works of Amrani et al. [4, 3] and Collberg et al. [25]. While all of these properties are important, they serve different purposes. For example, we want all transformation categories to keep the behavioral equivalence intact and to terminate for every input. Therefore, those properties are of lesser interest when analyzing the differences between the presented transformation categories. Nevertheless, we have to prove them for each transformation.

Behavior Preserving: A transformation category is considered to be “behavior preserving” if its application does never violate the behavioral equivalence (or at least the tolerable behavioral equivalence) of the input and the output model under the given selection S.

Determines: We consider a transformation category to determine if it always performs the same modifications for the same input model. In other words, it does not employ any randomness regarding the modification of the model.

Requirements: The requirements property is an artificial collection of features that the category may exploit or require. The features considered are:

- Discrete: The category does only work on discrete values (but is not restricted to whole numbers).
- Intervals: The category works on intervals that represent the domain of certain values.
- Time/External: The category makes use of external factors that may vary the evaluation result. The main external factor considered in this thesis is time.
3.2 Classification

- **Types:** The category has to know the types of all values and references that are present in the constraint. Depending on the typing system, this might not be equivalent to the requirement of a concrete type. For example, it might be enough to know that the reference evaluates to a numerical value.

- **User:** The category may require user interaction. Requiring user interaction is not equivalent with the category needing user guidance while it is applied but rather require an initial configuration that can not reasonably be generated automatically. One example for such a configuration is a tolerance factor by which a specific function is allowed to be distorted.

These features may assist in selecting specific transformation categories. As an example, it would not make any sense to select a category that can only operate on intervals if the modeling environment does not support them.

**Reversible:** A transformation category is “reversible” if its modification can be reversed without any or with relatively little guessing. Alternatively, we consider it reversible if it does not have to be reversed at all to grasp the meaning of the source. One example of a trivially reversible category is applying a conventional encryption method while providing the required decryption key (cf. Transformation Category 3). An example for a transformation category that does not need to be reversed at all is one that reorders components in a formula (e.g., it transforms \( a + b \) to \( b + a \) based on the commutative property, cf. Transformation Category 7). This definition of the term “reversible” is a weaker version of the “resilience” definition by Collberg et al. [25].

**Scope:** The scope categorizes the specific subphase(s) (cf. Subsection 3.1.3) in which the transformation category is to be applied: i) outer, ii) inner, iii) structure, and iv) cleanup represented by the numbers 1 to 4 respectively. Additionally, it specifies if the categories application is:

- **Local:** The category operates on a single constraint without looking at any other information from the model.

- **Global:** The category modifies multiple constraints, or requires information outside of the current constraint (e.g., if one of its references is referenced by another constraint).

**Termination:** Specifies whether the transformation category itself halts for every input after a finite amount of time. For the analysis, this only refers to the application of the respective transformation category and not to the transformation process as a whole (cf. Section 3.5). We expect all categories to terminate.

**Traceability:** The traceability property refers to the logging capabilities of the transformation category. In other words: if and how well the modifications of a given category can be logged to undo and redo its effects. In contrast to, e.g., Bergmann et al. (cf. [12]), we do not focus on bidirectional synchronization by tracing because we deal with one-way transformations.

However, besides the seven properties described, others can be analyzed as well. Some of them are discussed further in Note 36/A below.

---

**Note 36/A:** Additional Properties.


**1. Potency:** The amount of obscurity added.
2. **Resilience**: The difficulty for an automatic deobfuscator to reverse the obfuscation.
3. **Stealth**: How well the obfuscated part blends within not-obfuscated code.
4. **Cost**: The computation overhead added to the obfuscated program.

We do not analyze those criteria for every transformation category because a lot of them focus on the removal of information. They add no obscurity and have close to perfect stealth (as long as removing the information does not break an existing pattern). Furthermore, they achieve negative costs — at least most of the time — by removing expressions from the model or pre-evaluating parts of them.

With the resilience being loosely covered by the “Reversible” property, we discuss the quality criteria alongside a category only if they are of interest.

### 3.3 Transformation Catalog

This section describes and analyzes the “transformation categories”. A transformation category groups arbitrarily many single transformations so that they may fulfill a common purpose. In other words, when describing and analyzing a transformation category like “inline definitions”, we do not make any statements about the number of required underlying transformations to fulfill this purpose. Therefore, every category may be viewed as an entire program of its own. To inline definitions, there may be multiple transformations for different reference types or values operating behind the scenes.

#### 3.3.1 Overview

Table 38/A lists all transformation categories that are discussed in this section. We have selected these categories based on the selection criteria described in Note 37/A below. Furthermore, we structure those categories according to their type of operation and analyze them based on their impact on the four constraints (cf. Subsection 1.2.3) as well as their properties which we described in Subsection 3.2.2. The execution of the categories is discussed in Section 3.5.

The sixteen transformation categories presented in Table 38/A have been chosen from a literature survey — conducted during the writing of this thesis — interwove with a brainstorming of the authors. While this yielded a broad number of possible transformation categories, covering all of them would exceed the scope of this thesis. Therefore, we have selected sixteen of these categories based on the following criteria:

- **Necessity**: Some transformation categories are baseline required to allow for any modification. Consider a hierarchical model (cf. Example 29/A) whose hierarchical structure should be changed (e.g., to remove structural information for unselected model elements). Modifying the hierarchy can not be achieved without transformation categories that operate on the hierarchy, like Transformation Category 15.

- **Intuition**: Some transformation categories are intuitive for a one-way transformation approach, as the evaluation of constants. If we evaluate $25 + 17$ to 42, it is impossible to recover the original formula as there are infinitely many terms that evaluate to the value 42 (consider $(25 + k) + (17 - k)$ for any $k \in \mathbb{R}$).
### 3.3 Transformation Catalog

<table>
<thead>
<tr>
<th>Transformation Category</th>
<th>General</th>
<th>Requires</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>General Transformations</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T.1 Value Distortion</td>
<td>● ●</td>
<td>○ ○</td>
<td>● ●</td>
</tr>
<tr>
<td>T.2 Function Approximation</td>
<td>○ ●</td>
<td>○ ○</td>
<td>○ ○</td>
</tr>
<tr>
<td>T.3 Conventional Encryption</td>
<td>○ ●</td>
<td>● ●</td>
<td></td>
</tr>
<tr>
<td><strong>Disguise of computing paths</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T.4 Arithmetic Simplification</td>
<td>● ●</td>
<td>○ ○</td>
<td>● ○</td>
</tr>
<tr>
<td>T.5 Inline Definitions</td>
<td>● ●</td>
<td>○</td>
<td>● ○</td>
</tr>
<tr>
<td>T.6 Strength Manipulation</td>
<td>○ ●</td>
<td>● ●</td>
<td>○ ○</td>
</tr>
<tr>
<td>T.7 Operation Reordering</td>
<td>● ●</td>
<td>● ●</td>
<td>○ ○</td>
</tr>
<tr>
<td>T.8 Noise Creation</td>
<td>○ ●</td>
<td>○ ○</td>
<td>○</td>
</tr>
<tr>
<td>T.9 Oracle Rewrite</td>
<td>○ ●</td>
<td>● ○</td>
<td>○ ●</td>
</tr>
<tr>
<td><strong>Disguise of the result</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T.10 Tailor Expressions</td>
<td>● ●</td>
<td>●</td>
<td>○ ●</td>
</tr>
<tr>
<td>T.11 Reduce Constants</td>
<td>● ●</td>
<td>●</td>
<td>○ ○</td>
</tr>
<tr>
<td><strong>Disguise of the structure</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T.12 Identifier Obfuscation</td>
<td>○ ●</td>
<td></td>
<td>● ○</td>
</tr>
<tr>
<td>T.13 Remove Orphaned</td>
<td>○ ●</td>
<td></td>
<td>● ○</td>
</tr>
<tr>
<td>T.14 Constraint Encapsulation</td>
<td>○ ●</td>
<td></td>
<td>● ●</td>
</tr>
<tr>
<td>T.15 Rebuild Hierarchy</td>
<td>○ ●</td>
<td></td>
<td>○ ○</td>
</tr>
<tr>
<td>T.16 Remove Implicit</td>
<td>○ ●</td>
<td></td>
<td>● ●</td>
</tr>
</tbody>
</table>

**Table 38/A: Interesting properties of transformation categories.**

- ●: Required/Fulfilled.
- ○: Can exploit/fulfill.
- Default: Does not fulfill/require.

**Transformation Category:** Groups transformations serving the same purpose.

- General Transformations on the facing page.
- Disguise of computing paths on page 42.
- Disguise of the result on page 53.
- Disguise of the structure on page 56.

**External Factor:** Factors outside of a models control.

**Model:** Set of model elements.

---

**Interesting Properties:** While not all transformation categories can be covered and thoroughly analyzed in this thesis, we selected some of them because they reveal interesting behavior or because they have interesting properties.

We use the following four groups to structure the sixteen categories according to their type of operation:

1. **General Transformations**
   Those transformation categories do not fit into one of the other groups. However, they are described as starting points for further discussions.

2. **Disguise of computing paths**
   This group contains all transformation categories that focus on hiding the concrete operations for the calculation of values.

3. **Disguise of the result**
   These categories focus on the removal of information about inferred values. For example, by removing information outside of the scope of external factors.

4. **Disguise of the structure**
   The last group covers transformation categories that focus on modifying the hierarchy of models. It is of lesser interest if the input and output models do not have any structure. However, some transformations are still of importance as they decouple the obfuscated model from its origin.

In Table 38/A we visualize the last three groups as vertical lines whereas the general transformations are identified as the first three transformation categories.
3.3.2 General Transformations

This group contains three transformation categories which do not fit into the other groups. They are neither necessary nor intuitive but are listed solely for the interesting properties they exploit (cf. Note 37/A).

Value Distortion [T.1]

The first transformation category to be explained is named “Value Distortion” and is one of two categories that requires a custom user configuration to work. Furthermore, it is one of two categories that uses the definition of tolerable behavioral equivalence as its application changes the behavior of a model.

Before the description of the category itself, we define the information required by the user configuration. For every variable, we allow for a value range to be defined that should be used instead. It is up to the user frontend to decide whether to accept relative or absolute values. Yet, the configuration used by the category requires an absolute value range. This value range has to be a subset of the correct value range. In other words, the behavior of the category is undefined if a variable which is assigned the constant 5 is configured to represent the value 11 — checking this inclusion constraint is left up to the user frontend.

Through the construction we define the impact of this configuration to be “tolerable” in the terms of tolerable behavioral equivalence. As an example, configuring the value \( \eta \) for a variable with the real domain \( \delta \) (with \( \eta \subseteq \delta \)), may result in an update for all comparator functions \( \text{comp}_e(x,y) \) for every directly and indirectly affected variable \( e \) so that it allows for the deviation in the results to be compensated (e.g., by propagating a relative tolerance factor).

While the category can be configured so that it may choose a random value range based on certain policies configured by the user, we only consider a deterministic variant. Using a fixed value range can reveal less information when multiple variants of the obfuscated model are sent to the same individual. Furthermore, tracing the modification by this category is easy by storing the creation of the new constraint \( v = (\eta) \). We do not require typing for we expect \( \eta \) to be a subset of the correct value range.

Based on this, we consider the modifications of the category to be irreversible using the following arguments:

- By overwriting existing constraints, even when overwriting 5 with 5, the adversary can not know the concrete nature of the original value. This is even more extreme when overwriting a formula like \( \sin(x) \) with \([-1..1] \) erasing the trigonometric origin.
- This is assisted by the fact that the category only affects unselected model elements which are obfuscated further by transformation categories like the identifier obfuscation.

In the following, we discuss the effect of the transformation category on the four major constraints:

- **CI.1** \( B_{A,S} \) has to be self-contained, without any reference to any \( a \in \mathbb{E}_A \).
  
  By creating a new constraint \( v = (\eta) \) this category does not produce an additional reference to any \( a \in \mathbb{E}_A \) but retains those already existing which are to be obfuscated by T.12 Identifier Obfuscation.
3.3 Transformation Catalog

- C/2 B_{A,S} contains all of the elements in S and none of those in E_A \ S.
  As this category does specifically focus on model elements e \in E_A \ S it does not remove any unselected element. Yet, by overwriting a variable it allows for the inlining transformation category to inline all uses of the variable, making it applicable to be removed by T.1.3 Remove Orphaned.

- C/3 B_{A,S} may allow no inference on any model element e \in E_A \ S.
  Combining the arguments for the irreversibility and the discussion of C/2, this category solely overwrites the assigned formula or value range information. Therefore, it reduces the element beyond oracle access.

- C/4 The models A and B_{A,S} have to be behavioral equivalent.
  By definition, this category satisfies the tolerable behavioral equivalence which is configured by the user through \eta.

Finally, this transformation category terminates for any configuration as checking for a user configuration and creating/overwriting the desired constraint v = \eta is both possible in finite time.

Function Approximation [T.2]
The goal of the function approximation transformation category is similar to the previous T.1 Value Distortion but instead uses other functions to approximate an existing function (cf. [23]). Because there exists a lot of research along with well-elaborated algorithms performing this task (cf. [16, 13]), we refrain from diving into implementation details.

Considering the required user configuration of this category, a relative tolerance received by the frontend suffices, because algorithms like the one presented by Beylkin and Monzón allow for an arbitrary accuracy [13].

Describing any unselected constraint (and recursively any part of a constraint) as a function with all references in the constraints (or the respective part) as parameters, we can exchange the function by an arbitrarily accurate function approximation — where the accuracy depends on the tolerance defined by the user.

Just like before with Transformation Category 1, we assume that — by construction — the user configuration is equivalent with extending the comparator functions \text{comp}_e(x, y) for all directly and indirectly affected variables e.

If the approach does determine depends on the concrete implementation chosen, nevertheless the final modification can be traced by storing the performed constraint modification in the tracing model (see Subsection 3.1.1). The reversibility depends on the concrete implementation. Using a Fourier transform, there exist many types of functions that can be fully recovered [39, 19]. However, using a tolerance factor \epsilon \neq 0 that allows or even enforces a non-exact approximation yields infinitely many possible functions, resulting in an effectively irreversible approximation.

The approximation may be used along with external factors by treating them just as any other parameter. Furthermore, uncertainty intervals can be incorporated by approximating the functions describing the lower and upper bounds separately while compensating for the tolerance value.
The models

We discuss the last transformation category of the general group only briefly, as it

The formal language defined in Subsection 2.2.2 allows for the definition of functions different from them known in normal maths. However, describing approximation algorithms for every possible function exceeds the scope of this thesis — the referenced implementations assume the effectiveness of conventional arithmetic.

Nevertheless, as long as the functions in use allow for any reasonable approximation algorithm, the effects of this transformation category remain the same.

Just like before, we briefly discuss the effects of the transformation category on the four constraints:

- **C1**) Bₐ,S has to be self-contained, without any reference to any a ∈ Eₐ.
  Because this category does neither create nor remove any constraint — or any model element — it does not introduce a new reference to any a ∈ Eₐ, as long as the approximation does not use A as a base model. However, using A as a base would serve no other purpose as the same information could be taken from Bₐ,S as part of the obfuscation (e.g., by using T.12 Identifier Obfuscation).

- **C2**) Bₐ,S contains all of the elements in S and none of those in Eₐ \ S.
  By neither adding nor removing model elements, this transformation category does not change the elements in Bₐ,S.

- **C3**) Bₐ,S may allow no inference on any model element e ∈ Eₐ \ S.
  By approximating the original function of an unselected constraint, this category hides potential crucial formulas by less critical variants that may even result in the complete diffusion of some critical components [35]. Depending on i) the concrete approach, ii) the maximum acceptable tolerance, and iii) the domains of all parameters this may even result in the reduction of a higher-order polynom to a constant or linear function.

- **C4**) The models A and Bₐ,S have to be behavioral equivalent.
  Just like with T.1 Value Distortion this constraint is satisfied by definition, through the tolerable behavioral equivalence configured by the user.

The termination criteria of this category depend on the approach chosen. Nevertheless, common approaches like fast Fourier transform are all executed in a finite amount of time.

### Conventional Encryption [T.3]

We discuss the last transformation category of the general group only briefly, as it does not alter the stored information at all but only modifies the representation. Its application requires support by either the solver or the modeling frontend, and it is only to be applied after the expressions of the target model are created in the third subphase.

For any remaining unselected model element with an assigned expression in the output model the expression representation is to be encrypted using an arbitrary encryption algorithm (cf. [60]). Depending on the structure of the output model, the necessary decryption key may be added to the expression field (e.g., by enriching the solver with a decrypt function decrypt(<key>, <ciphertext>)) or stored in a meta field of the model format.

Depending on the encryption algorithm used, the transformation category can determine, while tracing is possible by storing the expression fields before and after the encryption alongside with the location of the decryption key. However,

---

**Note 41A:** Application of existing implementations.

**Subsection 2.2.2 on page 18.**

**Transformation Category:** Groups transformations serving the same purpose.

**Constraints on page 4.**

**Bₐ,S:** Model B build from A with selection S.

**Model Element:** Uniquely identified element of a model.

---


**Tolerable Behavioral Equivalence:** All variables in common evaluate to the same value (within a threshold) for all external factors.
because at least the solver has to decrypt the ciphertext — or has to receive an already decrypted variant — the category is baseline reversible no matter how the decryption instructions are transmitted.

Therefore, we do not discuss each of the four constraints separately: The category does not change the model elements in the model and only raises the obscurity of the representation.

Albeit the transformation removes no information, it may still be of interest in the case of automated attacks as it can distort their pattern matchings that check for specific parts of an expression.

Its potency (cf. Note 36/A) is very high as the ciphertext should not resemble the expression language. However, this implies very low stealth combined with negligible resilience. While automated attacks may be unable to pattern match the ciphertext directly, as soon as they implement similar deciphering algorithms as the solver or the frontend, they can read the original encrypted expression without any problem.

It can be argued that it would be possible to legally forbid deciphering using a proprietary cryptographic algorithm. While this would make (legal) reversibility practically impossible, it defies the overall purpose of the one-way transformation approach. If we allow for a proprietary format that the model might be stored in, revealing only the selected model elements to the user, there would be no need to remove any information. Additionally, there may be several reasons why just legally prohibiting decryption may not be enough:

1. It may be impossible to check if the data was decrypted nonetheless.
2. Depending on the country and the domain, the company may be baseline required to include a backdoor [75].
3. Even if prohibiting the decryption is admissible, there are no benefits in including data that would be removed by the obfuscation. Especially, as it ensures behavioral equivalence in correspondence to the selected elements.

### 3.3.3 Disguise of Computing Paths

Transformation categories of this group hide details about the computation of values. Therefore, for an inferred result of \( x \), these categories obfuscate parts of the required calculations to receive \( x \). The simplest form of this obfuscation is evaluating the equations beforehand. While we cannot evaluate an equation like \( 13 + k - 4 \) altogether, if \( k \) is unknown, we can still evaluate \( 13 - 4 \) and therefore change this term to \( 9 + k \). This pre-evaluation is performed by T.4 Arithmetic Simplification.

Some categories are described in pseudocode if the description adds benefit to the discussion. The definition of this pseudocode is explained in Appendix B. Nevertheless, it should be perfectly understandable on its own, using the appendix only as a reference in case of any ambiguity.

**Arithmetic Simplification** [T.4]

This transformation category does evaluate an expression for constant values where the result is neither directly nor indirectly dependent on external factors. If default maths apply, \( 3 + 2 - 4 \) evaluates to 1. For the explanation, we do assume that the expressions are represented as an Abstract Syntax Tree (AST) that integrates any precedence rules and that this AST is fully typed. That means each node in the AST is decorated with type information about the result.
Those requirements stem from the origin that we used in deriving this category: the constant folding compile optimization [26, Chapter 9].

**Requires:** Nonempty AST $a$ of an unselected constraint $c$.

**Requires:** Node $n$ of known function $f$ with $k$ constant leafs $l_1, \ldots, l_k$.

**Requires:** $f$ accepting the given signature $l_1, \ldots, l_k$.

**Requires:** $f(l_1, \ldots, l_k)$ only produces known references.

**Assures:** Replaces $n$ and $l_i$ with the result of $f(l_1, \ldots, l_k)$.

The pseudocode description of this behavior is as follows:

```
Function transform:
    Input: Node n of AST a.
    l ← (l_1, ..., l_k) ← children(n);
    if l are constant leaf and n has known function as f and
       f has signature(l):
       remove l_1, ..., l_k from a;
       replace n in a with node(f(l_1, ..., l_k));
    else:
       forall l_i ∈ l_1, ..., l_k do transform(l_i);
```

An example is depicted in Figure 43/A.

As this transformation category does only operate on a single constraint, it is considered to be local. By restricting the category to operate on known functions as defined in Subsection 2.2.2, the transformation determines by definition as the order of the children determines as well. Typing is required to identify the applicability of the function in question. We can trace its application, by recording the constraint before and after the arithmetic simplification.

Even though the definition of T4 Arithmetic Simplification restricts itself to unselected constraints, the classification allows it to be executed in the outer and in the inner obfuscation subphase (see Subsection 3.1.3). This is important in the combination with T5 Inline Definitions.

Whenever a definition $δ$ is inlined into a selected constraint $c$, we allow the arithmetic simplification to be executed on the largest subtree of the AST of $c$ which contains $δ$, an operator receiving $δ$ as a parameter, and no other reference.

Consider the constraint $a = b + 13 \times 2$ where it is known, that $b$ is $16$. After inlining $b = 16$ the arithmetic simplification category is executed on the subtree for $16 + 13 \times 2$ resulting in the updated selected constraint $a = 42$.

We do not differentiate between the two usages of the arithmetic simplifications as the arguments remain the same.

The following points discuss the influence of the transformation category on the four constraints:

- **C1)** $B_{A,S}$ has to be self-contained, without any reference to any $a \in E_A$.
  
  As exactly the references from the model are known (cf. Subsection 2.2.2), this transformation is unable by construction to produce any reference to $a \in E_A$ that is left unchanged by the identifier obfuscation.

- **C2)** $B_{A,S}$ contains all of the elements in $S$ and none of those in $E_A \setminus S$.
  
  First, this transformation category does only operate on a single constraint at a time. Furthermore, it does not perform any structural modification. Therefore, it can not alter the elements that are contained in $B$. 

---

[26] Keith Cooper et al.
Engineering a Compiler.

Transformation Category: Groups transformations serving the same purpose.

Transformation Category 4:
Arithmetic Simplification.

<table>
<thead>
<tr>
<th>Determines</th>
<th>Traceable</th>
<th>Reversible</th>
<th>Time</th>
<th>Intervals</th>
<th>Types</th>
<th>Discrete</th>
<th>User</th>
<th>Local</th>
<th>Phase</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 43/A: Example for the arithmetic simplification.

AST: Abstract Syntax Tree
Normalized Constraint: A constraint following the predefined syntax rules.
Subsection 2.2.2 on page 19.

Note 43/A: Arithmetic simplification in the inner subphase.
Subsection 3.1.3 on page 32.
The models following the predefined syntax rules. Behavioral Equivalence can allow the user to specify custom policies to exchange or forbid certain functions, yielding the responsibility to the user. If necessary, the implementation can allow the user to specify custom policies to exchange or forbid certain functions, yielding the responsibility to the user.

Behavioral Equivalence: All variables in common evaluate to the same value for all external factors.

Transformation Category: Groups transformations serving the same purpose.

Normalized Model: Flat assumption-free mapping of constraints to their origins.

AST: Abstract Syntax Tree

Normalized Constraint: A constraint following the predefined syntax rules.

Example 44/A shall illustrate the application of a transformation category.

For this example, we know only two functions + and − which are defined as the mathematical operations of addition and subtraction:

\[
\{ +: \mathbb{R}^2 \rightarrow \mathbb{R}, (a, b) \mapsto a + b, \quad -: \mathbb{R}^2 \rightarrow \mathbb{R}, (a, b) \mapsto a - b \}
\]

Furthermore, we consider \(x\) and \(y\) to be references in the statement: \(x = 13 + 5 - 2 \ast y\). The corresponding AST can be found in Figure 44/A.

For this equation, the steps of the algorithm would be as follows:

1. Receiving the children from the root node: \(l_1 = x\) and \(l_2 = -\) (for this explanation, we refer to the nodes with the values they contain). As \(l_1\) does not satisfy the constant predicate, the transformation is repeated for both children.

2. For \(l_1\), the value itself is not a known function and is skipped.

3. For \(l_2\) neither of its children \(l_1' = +\) and \(l_2' = \ast\) are a function. The transformation continues with \(l_1'\).

4. For \(l_1'\), all children \(l_1'' = 13\) and \(l_2'' = 5\) are constant and fulfill the leaf predicate. Furthermore, + is defined exactly for two parameter (which have to be typed at least as an element of \(\mathbb{R}\) in this example). Therefore 13 and 5 are removed, and + is replaced by the result of \(+13, 5\) = 18.
5. For $t'_2$, the right child $y$ does not satisfy the constant predicate. Therefore, the transformation is repeated for $2$ and $y$ but does not change anything, as none of them are functions.

For now, the definition deals only with a certain subset of functions and may be insufficient for modeling languages which allow for a wider set of functions to be used. Therefore, Note 45/A widens the range of applicable function to include arbitrary functions while explaining potential problems that come with the generalization.

The stated evaluation conditions require all parameters to be constants. Yet, there may be a function that receives a reference (e.g., as output parameter) or functions that can operate on intervals. Consider a ceiling function that receives a value that is known to lie in the closed interval $[3.8, 13.2]$. For $\text{ceil}([3.8..13.2])$, we can safely evaluate this to $[4..14]$, as this is the resulting range a value $x \in [3.8, 13.2]$ has to lie in if transformed by the ceiling function. As an example for allowed references, consider an array-function: $\text{array}(\langle a \rangle, \langle n \rangle)$ which receives an array-reference $a$ and an index $n$ and evaluates to the respective element no matter if it is a constant or not.

While dealing with $\text{ceil}$ is easy, we can not simply allow and evaluate any function. Consider a function like $\text{assert}(\langle x \rangle)$, which produces another constraint that assures $x$ is fulfilled. With this modification, the arithmetic simplification would no longer be local and can invalidate the constraints — for example, by enforcing an element outside of the selection — if the selection definition is not expanded accordingly.

To cope with all these possibilities, one may create a predicate which checks if $f$ can be applied safely under the given conditions. While defining such a predicate exceeds the scope of this thesis and might even be impossible for user-defined functions, it would even allow to handle constant or identity functions (e.g., $f(x) = 1$).

### Inline Definitions [T.5]

This transformation category is executed in the outer and inner obfuscation subphases. In the outer phase, only unselected normalized constraints are considered as inlining targets, in the inner phase it is allowed to inline definitions from unselected to selected constraints, but never inlining the other way around.

This category benefits from the identifier references of the second phase (cf. Example 32/A) to determine if the result evaluated for any given reference can be potentially influenced further.

In order to ensure that we can safely inline a value, we have to be sure that we loose no impact (cf. Transformation Category 13). Example 45/A below showcases potential problems in identifying impacts.

Consider $A = B + 2$ with the two constraints $B = [5..6]$ and $B = 5$. If we would use only $B = [5..6]$, inlining $B$ would resolve in $A = [5..6] + 2$, erasing any trace that the correct constraint would have been $A = 5 + 2$.

However, in this situation, we would expect the solver to infer the bound 5 for $B$. The problem is, we can not trust that the closest bound is not impacted further — especially if external factors are involved. If we would exchange $B = 5$ with $B = \text{if } T \geq \text{Jan2022 then 5 else 6}$ the closest bound would be “5 or 6". However, inlining this information would erase the concrete values that $B$ can receive, depending on the external factor time.
To cope with the problem of impact identification is easy if the expression language provides a way to retain required impacts. That is, as closest bounds inferred by the solver are never wrong by definition, we could inline a bound \( r = \beta \) keeping the effects of \( r \) intact with \( \beta_r \). However, as soon as other categories work on the inlined value, we would have to modify the effect of the impact.

If the language provides no mechanism of retaining the impact after inline, we have to identify the impacts remaining as described with T.13 Remove Orphaned or rely on certain restrictions. For example, we could make use of the exclusion of contradicting constraints. Hence, as soon as a constant value is inferred as the closest bound for a variable without the usage of external factors, we can assume it is unproblematic for inlining.

Suppose the modeling language allows for the definition of user-defined functions or has known functions (cf. Subsection 2.2.2) applicable for inlining. In that case, we could use the category similar to the inlining of methods in compiler construction [21, 26].

Such use-cases are common. In general, we could even use expressions to describe the bounds of a reference — this is what the Iris-solver does — which would allow accounting for the influence of external factors that can be represented in expressions. However, such discussions exceed the scope of this thesis and are thoroughly discussed by other researchers [21, 26].

As with the previous T.4 Arithmetic Simplification, we do assume that all constraints are represented as an Abstract Syntax Tree (AST) and that references are resolved and identified as such (cf. Subsection 2.2.3). Those requirements stem from the origin of this transformation: the constant propagation compile optimization [26, Chapter 9].

For each selected and unselected reference \( r \) in each unselected constraint \( c \) in the outer subphase and for each unselected reference \( r \) in each selected constraint \( c \) in the inner subphase. If there is a value \( v \) for \( r \) that can be inferred without removing any necessary impact, replace \( r \) in \( c \) with \( v \).

Although we could — in theory — only inline some values and pick those at random, we can remove more abstraction knowledge by using a determining version of inline definitions. Tracing is possible, by recording the constraint before and after the respective inline operation.

If one restricts the approach to only apply inline definitions, the category may not be necessary to reverse. That is, while it conceals information about the origin of values it does not remove the formulas the inlined value is based on. However, by inlining all the direct effects of such a formula, this category can make them obsolete for the behavior so they get removed by T.13 Remove Orphaned. Furthermore, the inlined values can be diluted further by applying T.4 Arithmetic Simplification.

It may be required to check for external factors and intervals to determine the impact. As this category has to look at other constraints to extract values, it is global.

We argue for the following effects on the four constraints by applying inline definitions:

- \( C_i \) \( B_{A,S} \) has to be self-contained, without any reference to any \( a \in E_A \).

Because this category only removes references, it does not reintroduce any reference to \( a \in E_A \).
However, this behavior may change if we allow the expressions to be inlined (cf. Note 46/A) to contain additional references. Nevertheless, all of them can be mapped by T.12 Identifier Obfuscation, leaving no reference to A when reconstructed with T.15 Rebuild Hierarchy.

- **C2**) \(B_{A,S}\) contains all of the elements in \(S\) and none of those in \(E_A \setminus S\).

  Inline definitions ever uses selected variables as a basis for inlining. Therefore, it only makes unselected normalized constraints applicable for removal by T.13 Remove Orphaned.

- **C3**) \(B_{A,S}\) may allow no inference on any model element \(e \in E_A \setminus S\).

  As discussed alongside the reversibility, the application of this category lays the ground work for removing inferable information by other transformation categories. Furthermore, inlining definition removes origin information.

- **C4**) The models \(A\) and \(B_{A,S}\) have to be behavioral equivalent.

  For we do only perform inlining whenever it does not break the impact of other normalized constraints it does not affect behavior.

Because the number of elements in normalized model is finite, and the impact analysis terminates (cf. T.13 Remove Orphaned), this category terminates as well.

**Strength Manipulation [T.6]**

The goal of strength manipulation is to obfuscate the calculation method used in an equation. It is derived from the strength reduction compiler optimization [27, 24] used to exchange expensive operations by less expensive equivalent ones [26, Chapter 8]. For the obfuscation, it is helpful to normalize the constraints (e.g., by exchanging all bit shifts with multiplications) allowing other transformation categories to proceed. Furthermore, it can increase obscurity by randomly exchanging operations which results in a low potency (cf. Note 36/A).

Consider a multiplication with an even number (e.g., \(x \times 6\)). This transformer is to swap any combination of an arithmetical left shift with a constant (e.g., \((x << 1) \times 3\)).

This category requires a set of predefined terminating transformers assigned to each operator. Those transformers \(t_i\) have to accept a function \(f\) and a list of parameters \(p_1, \ldots, p_k\) and produce another function \(o\) including a set of transformed parameters \(q_1, \ldots, q_l\) that is equivalent in terms of the language semantics. For example, this means, that \(f(p_1, \ldots, p_k) = o(q_1, \ldots, q_l)\). Yet, any \(t_i\) is allowed to return nothing, if it is incapable of transforming the given input vector \((f, p_1, \ldots, p_k)\). Example 47/A illustrates a transformer.

Assume a system with only two known operators: multiplication, denoted by \(*\) and an arithmetical left shift denoted by \(<<\). This transformer is to swap any arithmetical left shift by a constant with an equivalent multiplication:

**Input:** Function \(f\) and parameters \(p_1, \ldots, p_k\)

- \(f\) is \(<<\) and \(k = 2\) and \(f\) has signature \((p_1, p_2)\) and \(p_2\) is finite and \(p_2 > 0\):
  - \(q_2 \leftarrow \text{node}(2)\);
  - for \(i \leftarrow 1\) to \(p_2\) step 1 do
    - \(q_2 \leftarrow \text{node}(*, 2, q_2)\);
  - return \((*, p_1, q_2)\);
- else: return nothing;

See Figure 47/A and Figure 48/A for a graphical example with the input expression \(45 + (2 + x) << 3\) given as an AST. This is transformed to the expression if called for the \(<<\) node: \(45 + (2 + x) \times (2 \times 2 \times 2)\). This can be evaluated further by a transformation category like T.4 Arithmetic Simplification. In more
3.3 Transformation Catalog

detail, the function is $\ll$, and it has precisely two \texttt{children} (parameter). Furthermore, we assume that the arithmetic left shift is defined for the given addition and the numeric value $3$. As $3$ is finite and $3 > 0$, we do not return nothing and build the new AST for the multiplication with the while-loop. In the end, we return the output vector $(*, 2 + x, 2 * 2 + 2)$.

We expect those transformer functions to be “totally correct” \cite{Sokowski1987}. Hence, we expect them to terminate for any possible input vector. However, they do not have to determine as there may be infinitely many viable output vectors.

In the following description, we use the phrase “assigned to” to refer to the set of assigned transformers for a function. As an example, consider the one in Example 47/4A. We say, this transformer is “assigned to” the $\ll$-operator.

\begin{algorithmic}
  \Require Nonempty AST $a$ of an unselected \textit{normalized constraint}.
  \Require Node $n$ with children $c_1, \ldots, c_k$.
  \Require Node $n$ of known function $f$.
  \Assure Replaces $n$ and $l_1$ with equivalent $b$ and $l_1'$.
  
  The pseudocode description of this behavior is as follows:

  \begin{algorithm}
    \begin{algorithmic}
      \Function{transform}
      \Input{Node $n$ of AST $a$.}
      \State $l \leftarrow \{l_1, \ldots, l_k\} \leftarrow \text{children}(n)$;
      \If{$n$ has known function as $f$ and $f$ has signature($1$)}
        \State $t \leftarrow \{t_1, \ldots, t_2\} \leftarrow \text{known functions assigned to } f$;
        \While{$t \neq \emptyset$}
          \State $t_l \leftarrow \text{random}(t)$;
          \State $r \leftarrow t_l(f, l_1, \ldots, l_k)$;
          \If{$r$ equals nothing}
            \State $t^\prime \leftarrow t \setminus t_l$;
          \Else:
            \State $(o, q_1, \ldots, q_t) \leftarrow r$;
            \State remove $l_1, \ldots, l_k$ from $a$;
            \State replace $n$ in $a$ with node($o, q_1, \ldots, q_l$);
            \State \textbf{forall} $q_i \in q_1, \ldots, q_t$ do transform($q_i$);
            \State \Return;
          \EndIf
        \EndWhile
      \Else:
        \State \textbf{forall} $l_i \in l_1, \ldots, l_k$ do transform($l_i$);
      \EndIf
    \EndFunction
  \end{algorithmic}
\end{algorithm}

Given the implementation, this category \textit{category} does not determine if $t$ contains more than one known function ($t > 1$) or the transformer assigned does not determine. However, if required, it can be implemented determining, by choosing $t_i \in t$ based on a fixed heuristic (e.g., ordering them and iterating from 1 to $t$) and enforcing determining transformers. The strength manipulation can be traced by recording the expression before and after its application.

In general, it is unnecessary to reverse a strength manipulation for several transformers. Replacing an arithmetic left shift by a multiplication does not change the formula in a way that can be argued to secure an IP. However, this argument changes if the transformation category is combined with other categories like T.4 \textit{Arithmetic Simplification} that may reduce the arithmetic left shift into the multiplication with a magic number. Furthermore, the \textit{reversibility} depends on the implemented transformers as they can hide a proprietary or indicating operator that should not be revealed. For example, a transform could remove all
usages of an operator that is mostly used in one domain only, which would allow further guesses for an adversary. *Typing* is required to assess the applicability of certain operators. Some transformers may be restricted to *discrete* values.

Applying the *category* has the following effects on the four major constraints:

- **C1)** $B_{A,S}$ *has to be self-contained, without any reference to any* $a \in E_A$.
  
  This category does neither add nor remove *normalized constraints* or any *model element*. Therefore, it does not add a new reference to $a \in E_A$ that is not obfuscated by $T_{12}$ *Identifier Obfuscation*.

- **C2)** $B_{A,S}$ *contains all of the elements in S and none of those in* $E_A \setminus S$.
  
  Because strength manipulation does only operate on single unselected constraints, it can not remove any selected information.

- **C3)** $B_{A,S}$ *may allow no inference on any model element* $e \in E_A \setminus S$.
  
  Using the arguments made before about removing proprietary or indicating operations, this category can aide the reduction of inferable information.

- **C4)** *The models A and B_A,S have to be behavioral equivalent.*
  
  By definition the transformer has to yield the transformed function call $o(q_1, \ldots, q_t)$ with $f(l_1, \ldots, l_k) = o(q_1, \ldots, q_t)$ which does not effect the *behavioral equivalence*.

The termination of this category is not obvious. The basis is set by i) the finite number of nodes in the AST, ii) a finite set of predefined terminating transformers, and iii) the strongly monotonic reduction of elements in $t$ as long as they are not applicable. However, nothing prohibits the existence of a transformer, replacing $x$ with $f(x)$. Applying this transformer would result in an infinite recursion stemming from the innermost "else"-case. Therefore, great care is to be taken defining the guards of these transformers — discussing those guards exceeds the scope of this thesis.

### Operation Reordering [T.7]

The operation reordering transformation category is required to reorder the AST representation of constraints into a standardized form to drastically reduce complexity on the patterns to be matched by other transformation categories. While we refrain from discussing a specific standard form (as it is implementation-specific), we assume its existence [42].

It should be easy to imagine algorithms that exchange the order of operations. Moreover, because this category does not remove any information, the discussion remains on an abstract level.

---

**For each selected and unselected normalized constraint exchange the operation order while preserving behavior into a standardized format (“normal form”).**

If the standard form is unambiguous, the category has to *determine* in order to fulfill it. The *traceability* is guaranteed by recording the constraint before and after the standardization. Depending on the rules of the standardization, the execution may require *typing, intervals*, and information about the *external factors*.

Because we only change the order of operations (e.g., “$2 + x = 5$” to “$2 + x - 5 = 0$”) there is nothing that has to be reversed, making the transformation category *reversible*. Nevertheless, by transforming all constraints into a standardized form, we can still hide some intent embedded in the original notation.
The noise creation category raises the obscurity and tries to raise the resilience of a constraint (cf. Note 5.3). There is countless research on the generation of noise to raise the obscurity (cf. [57]). Therefore, we stick with an abstract description.

One of the simplest versions of noise creation is known as “dead code insertion” [9]. Inserting “dead code” would translate into the creation of constraints that do not affect the behavior of a model. For example, by replacing the insertion of literal dead code in expressions. For example, by replacing

\[ 3 + 2 = 5 \]

with

\[ 3 + \text{if } 14 = 5 \text{ then } 42 \]

or

\[ 3 + 2 \text{ with } 3 + 14 = 5 \text{ then } 42 \text{ else } 2 \]

Because increasing the obscurity works somewhat contrary to categories like Arithmetic Simplification or T.3 Remove Orphaned, we have to take some precautions to avoid losing the modifications of this category in multiphases. While the execution describes the T.3 Remove Orphaned, we make use of tainting to solve the other problems. That is, we increase the noise to raise the obscurity of a program (cf. [57]). Therefore, we stick with an abstract description.

It is possible to extend the effects of this transformation category to the structure of a model. For example, by creating passive model elements or splitting a block in multiple blocks. However, while this does not increase the obscurity, it does not reveal additional interesting properties. Therefore, we exclude the discussion in this thesis and defer it to potential future work in Section 5.3.

Conventional Encryption [T.3]

One of the simplest versions of noise creation is known as “dead code insertion” [9]. Inserting “dead code” would translate into the creation of constraints that do not affect the behavior of a model. For example, by replacing

\[ 3 + 2 = 5 \]

with

\[ 3 + \text{if } 14 = 5 \text{ then } 42 \]

or

\[ 3 + 2 \text{ with } 3 + 14 = 5 \text{ then } 42 \text{ else } 2 \]

Because increasing the obscurity works somewhat contrary to categories like Arithmetic Simplification or T.3 Remove Orphaned, we have to take some precautions to avoid losing the modifications of this category in multiphases. While the execution describes the T.3 Remove Orphaned, we make use of tainting to solve the other problems. That is, we increase the noise to raise the obscurity of a program (cf. [57]). Therefore, we stick with an abstract description.

It is possible to extend the effects of this transformation category to the structure of a model. For example, by creating passive model elements or splitting a block in multiple blocks. However, while this does not increase the obscurity, it does not reveal additional interesting properties. Therefore, we exclude the discussion in this thesis and defer it to potential future work in Section 5.3.
Using T.8 Noise Creation has the following effects on the four constraints:

- **C1** \( B_{A,S} \) has to be self-contained, without any reference to any \( a \in E_A \).
  
  The noise generation does create new constraints that may contain new references. Still, as all of them are changed by T.1 Identifier Obfuscation so they no longer reference any \( a \in E_A \).

- **C2** \( B_{A,S} \) contains all of the elements in \( S \) and none of those in \( E_A \setminus S \).
  
  This category makes no changes to selected constraints. Newly generated model element are create uniquely so they are not part of \( E_A \setminus S \).

- **C3** \( B_{A,S} \) may allow no inference on any model element \( e \in E_A \setminus S \).
  
  While raising the resilience, the effect on inferenceable information by a human or even automated attacker may be limited. Depending on the stealth of the noise generator, removing the added noise may even be trivial. Nevertheless, the added noise helps impede the effectiveness of automated attacks by targeting their pattern matching. Furthermore, noise can very well obscure intent.

- **C4** The models \( A \) and \( B_{A,S} \) have to be behavioral equivalent.
  
  By definition the generated noise does not interfere with the behavior.

We argued that this category must not determine as there is an infinite amount of noise to be added. However, in practice, it can be forced to terminate. This termination can be achieved by generating a random number first and then creating as much additional normalized constraints. By choosing an arbitrarily large yet finite value, we consider this category to be terminating.

### Oracle Rewrite [T.9]

The goal of an oracle rewrite is to cope with edge cases in which a function is discrete and called with a finite amount of input vectors but can not be obfuscated any further. It requires at least one of the references to be discrete and evaluates the function for all possible inputs for the discrete reference to remove any information on the complete implementation. For an input like \( x - 2 \) with \( x \in \mathbb{N} \) and \( 5 \leq x \leq 7 \) we do expect an output comparable to: if \( x = 5 \) then 3 else if \( x = 6 \) then 4 else 5. The problem of this approach is that if there are a lot of possible values for \( x \) (or if there are multiple discrete variables), it is possible to guess the underlying function with a pretty high probability — albeit there may be infinitely many polynomials in theory, for real-world use it seems to be highly unlikely for them to have a degree of, e.g., a hundred thousand.

However, this can not be helped. It is impossible to obfuscate a function representation further than oracle access if its input/output behavior is to be retained [64, 10].

Comparing T.4 Arithmetic Simplification with T.9 Oracle Rewrite, there are some important things to be made. The expressions evaluable by oracle rewrite are a superset of those of the arithmetic simplification. If all values are constants, they are — at the same time — discrete values of a finite domain defined by only the values themselves. For an expression like \( \text{floor}(3.2) \) — with \( \text{floor} \) being the floor function known from mathematics — oracle rewrite would produce the same result as the arithmetic simplification: 3.
3.3 Transformation Catalog

**Transformation Category:**
Groups transformations serving the same purpose.

**AST:** Abstract Syntax Tree

**Transformation Category 9:**
Oracle Rewrite.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Requires:** Nonempty AST \( a \) of element \( e \) with \( e \not\in S \).

**Requires:** Node \( n \) of known function \( f \) with \( k \) children \( l_1, \ldots, l_k \).

**Requires:** \( f \) accepting the given signature \( l_1, \ldots, l_k \).

**Requires:** At least one \( l_1, \ldots, l_k \) is discrete with finite domain.

**Requires:** \( f(l_1, \ldots, l_k) \) only produces known references.

**Assures:** Replaces \( n \) and \( l_i \) with an oracle for all possible inputs vectors.

The pseudocode description is as follows:

**Function transform:**

**Input:** Node \( n \) of AST \( a \).

\[
\begin{align*}
l &\leftarrow (l_1, \ldots, l_k) \leftarrow \text{children}(n); \\
l' &\leftarrow (l'_1, \ldots, l'_k) \leftarrow \text{filter } l \text{ has finite domain}; \\
\text{if } j \geq 1 \text{ and } n \text{ has known function as } f \text{ and } f \text{ has signature}(l) : \\
&\quad \text{remove } l_1, \ldots, l_k \text{ from } a; \\
&\quad \text{oracle } \leftarrow \emptyset; \\
&\quad \text{for } i \in \text{domain}(l'_1) \times \ldots \times \text{domain}(l'_k) \text{ do} \\
&\quad \quad \text{oracle } \leftarrow \text{if } i = l' \text{ then } \text{"evaluate } f \text{" else oracle}; \\
&\quad \text{replace } n \text{ in } a \text{ with oracle.} \\
\text{else:} \\
&\quad \text{forall } l_i \in l_1, \ldots, l_k \text{ do } \text{transform}(l_i);
\end{align*}
\]

As this transformation category does only modify a single constraint, it is considered to be local. It can be traced by recording the expression before and after the oracle. The reversibility depends on the number of cases produced by the oracle. While irreversible in theory, generating an oracle for \( f(x) = x \) with \( x \in \mathbb{Z} \) being restricted by \(-10 000 \leq x \leq 12 000\) should be reversible to \( f \) as all other theoretically possible functions are highly unlikely. However, for smaller evaluation points and with smaller \(|k - j|\) the unlikeliness of the other functions dwindles.

The oracle requires typing to values from the domain and check the applicability of \( f \). Naturally, it requires discrete values for the for-loop to terminate.

We discuss the effects of an oracle regarding all four constraints:

- C(i) \( B_{A,S} \) has to be self-contained, without any reference to any \( a \in E_A \).

By merely pre-evaluating function calls, the oracle rewrite does not add any additional references. This goes along with the arguments made for the arithmetic simplification category.
3. One-Way Transformation

- **C2)** \( B_{A,S} \) contains all of the elements in \( S \) and none of those in \( E_A \setminus S \).
  Just like T.4 Arithmetic Simplification, this category does only operate on a single constraint at a time and does not perform any structural modifications. It can not alter the elements contained in \( B \).

- **C3)** \( B_{A,S} \) may allow no inference on any model element \( e \in E_A \setminus S \).
  This constraint is problematic as the whole reason for applying this transformation category is for functions which can not be evaluated/obfuscated further. Nevertheless, as is discussed with the limitations in Section 3.4, this is arguably the best we can do in some cases.

- **C4)** The models \( A \) and \( B_{A,S} \) have to be behavioral equivalent.
  The evaluation of \( f \) for every possible input vector creates a map of all possible input/output mappings. Because \( f \) has to determine, the pair-representation \( f(1) \mapsto 2, f(3) \mapsto 12 \) is another — yet equivalent — description for the same function. Therefore, the behavioral equivalence remains unharmed.

This category terminates. The finite AST limits the number of nodes to be visited. The filter runs in a finite time, as does the pre-evaluation of \( f \) — because the calculation of \( f \) itself terminates — and therefore the construction of the oracle based on the finite amount of elements in \( i \in \text{domain}(l'_i) \times \ldots \times \text{domain}(l'_i) \) guaranteed by the finite number of elements in each domain.

### 3.3.4 Disguise of the Result

Transformation categories of this group hide details about the results themselves by exploiting several domain limitations of external or internal factors (i.e., variables). This extends T.9 Oracle Rewrite for non-discrete values and applies even for infinite domains (e.g., \([3, \infty)\)).

**Tailor Expressions [T.10]**

The goal of the tailor expression transformation category is to tailor functions to the domains of input vectors (like external factors) explicitly.

Consider an expression \( \text{linear}(x, 0, 5, 10, 100) \) which interpolates a result for the value of \( x \) between \((0, 5) \) and \((10, 100) \). For \( x = 8 \) this expression evaluates to \( 81 \). If \( x \) is limited to \( x \in [0, 8] \), we can tailor the given linear-expression to \( \text{linear}(x, 0, 5, 8, 81) \), hiding calculation details outside of the domain impeding extrapolation.

As another example, consider the Iris-expression:

\[
\text{if } T \geq \text{Jan2020} \& T \leq \text{Jan2022} \text{ then } 5 \text{ else } 0
\]

\( T \) refers to the external time that we can not control, but which is fixed for the obfuscation to lie in between January 2021 and December 2023. Because of this, we can exchange \( T \geq \text{Jan2020} \) with \( T \geq \text{Jan2021} \) hiding information outside of the domain of external factors:

\[
\text{if } T \geq \text{Jan2021} \& T \leq \text{Jan2022} \text{ then } 5 \text{ else } 0
\]

Considering the external factor in Iris it is possible to bind \( T \) to another time locally. This does not impede the tailoring in general — it may even benefit the process — however, it requires greater care when tailoring an expression.

Using the example in Figure 54/A, we assume the global time is restricted to lie between January 2021 and December 2023. If we just look at “a”, tailor expression would presume the replacement of \( \text{Jan2020} \) by \( \text{Jan2021} \) resulting in a wrong evaluation for \( b \) which uses the local time dependence to access \( a \) in the time interval \([\text{Aug2020..Jul2023}]\).
Hence, we can not tailor the expression without taking into account all potential local bindings for \( T \). For this discussion, we address this issue by using the solver to infer the closest domain that contains all possible values for the local binding of \( T \). In this example, the closest domain would be the union of both intervals: \([\text{Aug}2020..\text{Dez}2023]\).

Another possibility would be to create clones of the accompanying constraints that represent each access using a local binding different from the global one and inline the enforced offset. This would allow tailoring each of the clones individually.

Similar to the strength manipulation category we require a predefined terminating tailoring function that is is assigned to each known function applicable for tailoring. We only allow one tailoring function as we would very much like it to determine. If it is incapable of tailoring the expression presented (for example, because the closest inferrable bounds are all infinite), the tailoring function returns the value \( \text{nothing} \).

The tailoring function receives the closest domain for references and \( \text{external factors} \) as inferred by the solver. We denote the domain information with \( \delta \).

The abstract behavior of this category is similar to that of T.6 Strength Manipulation category albeit their intent is different. Using the tailor expressions category we change the information effectively retained in an expression — nevertheless, both categories share similar problems.
Assuming the tailoring functions determine the property propagates to the category. We trace by storing the expression before and after the modification. In general, this transformation is irreversible albeit external semantics by the modeling domain may allow educated guesses. If we know that the function is linear for \([a, b]\) and \([c, d]\) with \(b < c\) it may be safe to guess the values for \([b, c]\) and use this information for extrapolate for values outside of \([a, d]\). Because we need the closest domain for all references and external factors in question, the category is considered to be global.

We argue for the following effects on the four constraints:

- **C1** \(B_{A,S}\) has to be self-contained, without any reference to any \(a \in E_A\).
  This category removes only parts of an expression, never creating any unknown reference. Therefore, it is incapable of invalidating this constraint.

- **C2** \(B_{A,S}\) contains all of the elements in \(S\) and none of those in \(E_A \setminus S\).
  Because this category does not perform any structural modification, it can not alter the elements contained in \(B\).

- **C3** \(B_{A,S}\) may allow no inference on any model element \(e \in E_A \setminus S\).
  As we have already argued with the reversibility of this category, its application does impede potential extrapolation.

- **C4** The models \(A\) and \(B_{A,S}\) have to be behavioral equivalent.
  The tailor expression category only removes information for input-vectors that lie outside of the obfuscation domain. Because the behavioral equivalence respects the domain of external factors, removing information outside of it does not affect the equivalence.

The proof of termination has the same problems encountered with T.6 Strength Manipulation. For example, if there is a tailoring function replacing \(x\) with \(f(x)\), it would result in an infinite recursion stemming from the innermost if-case. Therefore, great care is required when constructing the tailoring functions so they do not exhibit non-terminating behavior.

**Reduce Constants [T.11]**

The goal of the reduce constant category extends on T.5 Inline Definitions. While the latter one restricts itself to inline only if we know that we do not lose any calculation information, reducing constant makes use of the fact that we know constraints outside of the selection are not subject to change.

For example, consider two constraints in the form \(x \geq 2\) and \(x \leq 5\) that require \(x\) to lie in the closed interval \([2, 5]\). If both constraints are outside of the selection and therefore are not be changed we can simply intersect the domain of \(x\) by their restriction. If \(x\) is currently \((-\infty, \infty)\), this would result in a new constraint: \(x = [2, 5]\). In other words: we replace the value for \(x\) with the closest domain inferred by the solver (as long as no external factors are in play, cf. Transformation Category 5). Afterwards, the value \(x\) is applicable for inlining. Inlining would not work if \(x \geq 2\) would stem from a selected constraint. If someone changes the constraint to \(x \geq 3\), the effects would no longer propagate to inlined values leaving them unchanged and potentially wrong — possibly resulting in contradicting constraints.
Transformation Category 11: Reduce Constants.

Determined: ○
Traceable: ○
Reversible: X
Time: ○
Intervals: ○
Types: ○
Discrete: X
User: ○
Global: X
Local: ○
Phase: 1, 2

Normalized Model: Flat assumption-free mapping of constraints to their origins.
Normalized Constraint: A constraint following the predefined syntax rules.

Transformation Category: Groups transformations serving the same purpose.
External Factor: Factors outside of a model's control

Constraints on page 4.
Subsection 2.2.3 on page 20.

Model Element: Uniquely identified element of a model.
Behavioral Equivalence: All variables in common evaluate to the same value for all external factors.
Huskied Model: One-way transformed model satisfying the four constraints.
Model: Set of model elements.

Subsection 2.2.3 on page 20.
Tracer/2 on page 30.

Note 56/A: Application notes.

For every unselected (outer subphase) and selected reference (inner subphase) \( r \) in the normalized model, infer the closest possible bound \( \beta_r \) using only unselected constraints. For each \( \beta_r \), produce a new unselected constraint \( r = (\beta_r) \).

Because there is only one closest possible \( \beta_r \) for each reference, we consider the category to determine. We can trace the effects by storing the newly generated constraints, that are — by construction — unable to create new contradictions. External factors are required as they can have a definite influence on the closest possible bounds. By considering all unselected and selected constraints, this category is global.

Due to the category not modifying the constraint in the normalized model we can summarize the effects on the four constraints based on the arguments before as they are not harmed.

The termination follows based on the finiteness of the normalized model and the solver terminating by definition. That is, we assume that if the solver terminates inferring bounds using the complete normalized model, it terminates as well using only the unselected part (cf. Subsection 2.2.3).

3.3.5 Disguise of the Structure

Transformation categories of this final group deal with the structural modification of a model, including the obfuscation of reference, hierarchy, and usage information. The most important of those categories is T.13 Remove Orphaned, which removes normalized constraints and model elements that are no longer required to ensure the behavioral equivalence of the Huskied Model.

Additionally, this phase is responsible for the recreation of a model based on the normalized model.

Identifier Obfuscation [T.12]

The goal of the identifier obfuscation transformation category is the exchange of identifiers with newly created ones so that it is impossible to reason about the original identifiers. This category populates the mapping function \( m \) allowed in the definition of the behavioral equivalence in Subsection 2.2.3 and singlehandedly ensures, that all references to the original model are removed.

We populate the mapping function \( m \) is part of the tracing model through Tracer/2, by writing mapping pairs \( \alpha \mapsto m \beta \).

While it is possible to execute this transformation category in a single pass at the start of the structure obfuscation subphase, it can be integrated into the concrete creation process of the output model with T.15 Rebuild Hierarchy. Therefore, it may be called for each reference encountered during the re-creation of the output model.

A modeling language may very well already use randomly created unique identifiers which reveal no semantics. Furthermore, they may even forbid aliases that could be used to transport some intent. However, even if that is the case, executing T.12 Identifier Obfuscation ensures randomly generated new unique identifiers that guarantee no backlinks to the original model.

To abstract away the point of application (cf. Note 56/A) we restrict the category to the obfuscation of a single unique identifier at a time. Nevertheless, if the category is applied before any other category of the third subphase it benefits form a identifier mapping as presented in Example 32/A.
For each reference $r$ encountered in the AST, we check first if the mapping function $m$ does already contain a mapping for $r$. If it does, we replace $r$ by $m(r)$, otherwise, we randomly generate a new unique identifier $r'$, store it in $m$ via $r \mapsto m(r')$ and replace $r$ with $r'$.

Relying on $m$ from the tracing model, we consider T.12 Identifier Obfuscation to be global and traceable by storing the normalized constraints before and after the application — alongside newly produced mappings — in the tracing model. Because there is no correlation (except the identity preserving part) between the original reference and the new one, this category is irreversable.

It should be noted that it is theoretically possible to create a determining variant of the identifier obfuscation. For example, by incrementing a counter for each newly encountered reference (cf. Note T.72). As long as the process itself does not allow to make any guesses about, e.g., the number of elements in the original model, the impact of this category remains the same.

Applying the identifier obfuscation on the four constraints has the following consequences:

- **C1)** $B_{A,S}$ has to be self-contained, without any reference to any $a \in E_A$.
  
  This category creates only new references that, by construction, cannot reference any $a \in E_A$.

- **C2)** $B_{A,S}$ contains all of the elements in $S$ and none of those in $E_A \setminus S$.
  
  This transformation category does neither remove nor add a new model element. Yet, it exchanges their unique identifiers.

- **C3)** $B_{A,S}$ may allow no inference on any model element $e \in E_A \setminus S$.
  
  By removing potential information introduced by aliases or the identifier generation process of the modeling language, this transformation category reduces the potentially inferrable information.

- **C4)** The models $A$ and $B_{A,S}$ have to be behavioral equivalent.
  
  As this category modifies only the identifiers but consistently exchanges all identifiers identity preserving, this has no effect on the behavioral equivalence.

Proving the termination for the single application of the identifier obfuscation is simple. Checking, generating, and mapping the old references to the new ones is possible in a finite amount of time.

**Remove Orphaned [T.13]**

This transformation category is critical for hiding unwanted information as it is responsible for the removal of normalized constraints that have no longer any influence on the behavior of the selected variables.

In the discussion below we use the phrase “constraint […] has no effect on the behavior of any selected variable”. Therefore, we require a way of identifying the effects that constraints have on the result inferred by the solver. The following example T.57A exemplifies a naive approach.

Let $N$ be the normalized model and $C$ be the set of all normalized constraints in $M$. Build an undirected graph $G = (V, E)$ with $V = C \cup \bigcup_{n \in E_M} E_n$ and edges, based on the following rules starting with $E = \emptyset$. For each $c \in C$ and for each reference $r$ in $c$ create an edge $E = E \cup (c, r)$.

For each selected $n \in E_N$, mark it and all reachable $v \in V$ inside of $G$. 

---

**Transformation Category 12:** Identifier Obfuscation.

- Determines: ☐
- Traceable: ☐
- Reversible: ☒
- Time: ☒
- Intervals: ☒
- Types: ☒
- Discrete: ☒
- User: ☒
- Global: ☒
- Local: ☒
- Phase: 3

**AST:** Abstract Syntax Tree

**Normalized Constraint:** A constraint following the predefined syntax rules.

**Transformation Category:** Groups transformations serving the same purpose.

**Model Element:** Uniquely identified element of a model.

**Model:** Set of model elements.

- ☐ Constraints on page 4.
- $B_{A,S}$: Model $B$ build from $A$ with selection $S$.

**Behavioral Equivalence:** All variables in common evaluate to the same value for all external factors.

**Variable:** Model element that receives a value by the solver.

**Example T.57A:** Naively identify the impact of constraints.

**Normalized Model:** Flat assumption-free mapping of constraints to their origins.

$\forall_{E_M}$: All variables in $E_M$. 

---

57
Every marked constraint $c \in C$ has an effect on the behavior of at least one of the selected $n \in V_E^N$.

However, this naive way would conclude that $X \geq 3$ has an effect on $X = [4..5]$ albeit it may be argued that this constraint has no further effect on the value of $X$. There is a lot of research on identifying the impact of constraints on each other, which exceeds the scope of this discussion [69, 80].

A simple and correct — albeit very ineffective — approach would be to solve the normalized model after having removed any combination of unselected constraints and mark the overall largest set of removable constraints which does not affect the behavior of the selection. All constraints marked are applicable for removal. We can improve on this approach in various ways. For example, by

- using the approach from Example 57/A first to reduce the number of constraints to be considered.
- check for certain normal forms in the constraints (e.g., assignments) and do not try them with every combination. They can still be removed if all constraints referencing them are removed with the costly approach.

No matter which approach we are using. We are, by definition, never removing tainted constraints.

Remove each unselected and untainted (cf. T.8 Noise Creation) normalized constraint in the normalized model that has no effect on the behavior of any selected variable.

Depending on the method we use to identify the impact of unselected constraint this method may determine. In general, it does not have to. Consider two redundant constraints $X \geq 3$ whereas either one of them has an effect, while the other one does not — the one to remove can be chosen randomly. Tracing can be achieved by storing the removed constraints in the tracing model. Because we remove the constraints completely, we consider the category to be irreversible.

Removing orphans, has the following consequences on the four constraints:

- **C1)** $B_{A,S}$ has to be self-contained, without any reference to any $a \in E_A$.
  - Because we only remove constraints we reduce the number of references to $a \in E_A$ at most.
- **C2** $B_{A,S}$ contains all of the elements in $S$ and none of those in $E_A \setminus S$.
  - As only unselected constraints are removed, nothing selected is affected.
- **C3** $B_{A,S}$ may allow no inference on any model element $e \in E_A \setminus S$.
  - In combination with other categories, like inline definitions, this category is key in impeding the inferenceable information on unselected elements.
- **C4** The models $A$ and $B_{A,S}$ have to be behavioral equivalent.
  - By construction, we only perform removals, that do not affect behavioral equivalence.

The termination of this approach depends on the algorithm used to identify the impact of constraints. However, because both example approaches described to terminate, we consider the category to be terminating.
Constraint Encapsulation [T.14]

This category works partially against the splitting of constraints described in Section 2.2. That is, the goal of this category is to raise obscurity right before T.15 Rebuild Hierarchy. As an example it may join two constraints $C_1$ and $C_2$ together to the new constraint $C_1 \land C_2$. Additionally, it may extract parts of an expression into new constraints. In the scope of inline this could imply the rewriting of "if $T \geq \text{Jan2020}$ then 3 else 2" as "if $A(T)$ then $B(T)$ else 2" with constraints "$A(T) = (T \geq \text{Jan2020})$" and "$B(T) = 3$".

We call those operations “join” and “split” respectively. They resemble the inline and outline code obfuscations mentioned by Balakrishnan et al. [8].

For the category to work, we assume:

- the modeling language has a constraint model element producing a constraint that equals the expression given.
- Just like with T.15 Rebuild Hierarchy, we require this to abstract away the problem of transforming constraints back to model elements.
- there is a logical “and” operator or a similar function.
- We write this operator as “&”. However, if the modeling language provides no way of joining two constraints in expressions, “join” can not be performed.
- every subtree inside an AST of a constraint may be replaced by a reference to a variable with an expression build from it without changing behavior.
- If this is not the case, we expect a predicate that can decide if extraction is possible for each subtree of each normalized constraint in each model.

In the following description of the category, we present two algorithms, one for split and join. The symbol $\rho$ represents the set of all available unique identifiers, provided and updated by the tracing model. We write “Create new constraint” and “Rebuild Hierarchy” together.

For the addition of another normalized constraint to the normalized model $N$.

### Split

Described as follows for unselected $a$:

**Function transform**:

- **Input**: Node $n$ of AST $a$.
- $l \leftarrow (l_1, \ldots, l_k)$ ← filter children($n$) is not leaf and not reference;
- if $k \geq 1$:
  - $s \leftarrow \text{subtree}(\text{random}(l))$;
  - $r \leftarrow \text{random}(\rho)$;
  - Create new constraint “$r = (s)$” in N;
  - replace $s$ in $a$ with $r$;
- else:
  - $\forall l_i \in l_1, \ldots, l_k$ do transform($l_i$);

### Join

For two randomly chosen unselected constraints $C_1$ and $C_2$. Remove them from N. Add the new constraint $C_1 \land C_2$ to N.

By choosing unique identifiers randomly, the argument if this category determines is similar to T.12 Identifier Obfuscation. The traceability is ensured by recording the removed and added constraints in the case of a “join” and recording the newly created constraint alongside the modifications in case of a “split”.

We consider T.14 Constraint Encapsulation as reversible because its application does only modify the presentation of the constraints and not the information stored inside the complete normalized model. However, applying the category is able to hide intent, similar to inline and outline methods (cf. [8]).

---

3. One-Way Transformation

Transformation Category: Constraint Encapsulation.

- Set: ○
- Traceable: ⊙
- Reversible: ●
- Time: ×
- Intervals: ×
- Types: ×
- Discrete: ×
- User: ×
- Global: ○
- Local: ○
- Phase: 3
The effect on the four constraints is minimal. While we do change the constraints in the normalized model it is to be argued that we neither remove any existing reference. By definition, we forbid constraint encapsulation to modify the behavior — if this can not be guaranteed, it is not to be applied.

For all modifications to the normalized model are terminating, the termination of this transformation category is based on the finiteness of the AST.

Additionally, the repeated application of split and join individually reaches a fixed point. For join, this is shown by the strictly monotonous reduction of unselected constraints in N until only one is left, which contains all originally unselected constraints. For split the termination is based on the fact that it always ignores the root node. That is, each split reduces the height of the subtree at least by one. Once reaching subtrees of height 1, split does no longer apply.

Nevertheless, both operations work against each other. That is, repeatedly outsourcing and joining constraints can increase the normalized model arbitrarily. For example, consider \( A \land B \land (B = (3 + 2)) \). Suppose, split extracts \( 3 + 2 \) to \( B = (3 + 2) \), leaving \( A \land B \). However, join can produce a new constraint \( A \land B \land (B = (3 + 2)) \) which in turn could be split again and so on. This is dealt with in Section 3.5.

### Rebuild Hierarchy [T.15]

The goal of the rebuild hierarchy category is to create the model based on the constraints in the normalized model. After its application, all further transformations of the cleanup phase operate on the generated model.

Consider the Iris hierarchy from Figure 60/A. If only Parent.A and Parent.A.X are selected, we would either have to create a fake block surrounding A or we can re-parent A (as nothing effected by A being a child of Parent is selected) making it a block without a parent.

As the process of recreating an output model depends on semantics of the modeling language (e.g., when to create a structural model element) and the specific constraint model element we describe the category on a very abstract level (see Subsection 4.1.1). We expect the following things:

1. The modeling language allows for a constraint model element that produces a constraint that equals the expression given.

   If this is not guaranteed, we have to ensure that each possible constraint can be represented with one or multiple model elements. This may include restricting the transformation categories, so this requirement can be ensured.

2. The modeling language has a variable model element.

   Fulfilling this requirement is necessary to express any output behavior. Therefore, we do only consider modeling languages that support variable model elements in the first place.

3. The models are hierarchical.

   If the modeling language does not have the concept of hierarchy, it is easy to adapt the abstract algorithm description by simply removing all checks and potential creations of structural model elements.
Start with an empty output model M. Store all potentially newly generated identifiers in the tracing model. For each constraint in the normalized model check its origin.

**Selected-Hierarchy.** If the constraint is selected, assure the original hierarchy for the origin model element is already in M. Do so by producing the structural model elements for each selected element in the hierarchy and producing a randomly identified placeholder for it if it is not.

**Unselected-Hierarchy.** If the constraint is unselected, no hierarchy has to be created. Instead, it is added to the uppermost hierarchy level.

**Constraint and Variables.** Create a new constraint model element — that maps the expression directly to the constraint — at the uppermost hierarchy level and set its expression to the constraint. Than, for each reference in the constraint, check if its mapped counterpart in the original model was selected. If so, assure its hierarchy (as before). If the reference is not mapped to a selected variable, use the uppermost hierarchy. Create a new variable model element using the unique identifier of the reference.

Because we choose new identifiers randomly, the category must not determine — the argument is similar to T.12 Identifier Obfuscation. However, all effects remain traceable by storing all newly generated elements in the tracing model. The category may allow for user interaction by accepting configurations if the hierarchical structures of the modeling language are ambiguous. Because this category finally creates the output model in question, the discussion about its effects on the four constraints is of special interest:

- **C1** $B_{A,S}$ has to be self-contained, without any reference to any $a \in E_A$.
  Assuming, T.12 Identifier Obfuscation already operated on the normalized model, all unique identifiers are newly generated. Therefore, the model does not reference or use any model element from A.

- **C2** $B_{A,S}$ contains all of the elements in $S$ and none of those in $E_A \setminus S$.
  The effect on the selected elements depends on the transformation categories employed before. If all of them remain in the normalized model — which is what we expect — this category ensures that all elements are present.
  However, the modeling language may not create a constraint for a model element that is selected but neither referenced nor assigned to an expression. To cope with this, we can manually ensure the existence of all selected model element by an extra pass over the set of selected model element, creating all missing elements.

- **C3** $B_{A,S}$ may allow no inference on any model element $e \in E_A \setminus S$.
  The major purpose of this category is to produce a model and not to reduce the inferable information. Therefore, it does not remove any information. On the contrary, it may even re-induce some information by creating hierarchical structures removed by the selection.
  Nonetheless, if the exact model structure is crucial for the modeling language syntax, this can not be helped (see Subsection 4.2.1).

- **C4** The models $A$ and $B_{A,S}$ have to be behavioral equivalent.
  By essentially just reformattting the normalized constraints to be set inside constraint model elements that re-generate the exact same normalized constraint, we do not change the behavioral equivalence. For implicit assumptions — for we require them to be explicitly expressible (cf. Subsection 2.2.2) — they can not introduce redundant or even contradicting effects.
Based on the finite hierarchy of the original model and the finiteness of the normalized model, this category terminates.

Remove Implicit [T.16]
The remove implicit transformation category is of use if any implicit assumptions remain in the model produced by T.15 Rebuild Hierarchy.

The job of this category depends completely on the implicit assumptions by the specific modeling language — if there are none or if they can not remain unchanged, this category is of no further use.

We do not give a detailed description of its formal behavior as the formalization allows for any kind of implicit assumption as long as it can be expressed explicitly (cf. Subsection /two.oldstyle./two.oldstyle./two.oldstyle). However, the algorithm resembles that of T.13 Remove Orphaned but focuses on the impact on the generated normalized model not on the impact on the inferred results.

Remove each model element — stemming from an unselected normalized constraint — in the model, whose effects are completely covered by implicit assumptions.

Similar to T.13 Remove Orphaned, there may be situations in which, for example, one of two redundant model elements may be chosen at random. Therefore, the category must not determine but can be traced by record the removed model elements in the tracing model. However, we only remove model elements that are already covered by implicit assumptions. As we do not remove any information from the model but rather hide information that could be used to identify what has been obfuscated, we consider this category to be reversible.

The category is global because we probably have to consider the impact of multiple model elements in combination with their hierarchy to determine if they are covered by implicit assumptions.

Removing the implicit assumptions has the following impact on the four constraints:

- **C1** $B_{A,S}$ has to be self-contained, without any reference to any $a \in E_A$.

  Because the model $B_{A,S}$ is already built by T.15 Rebuild Hierarchy and because we do only remove model elements, we can not re-introduce references to $a \in E_A$.

- **C2** $B_{A,S}$ contains all of the elements in $S$ and none of those in $E_A \setminus S$.

  We do only remove model elements stemming from unselected normalized constraints. Therefore, we never remove the elements from the selection.

- **C3** $B_{A,S}$ may allow no inference on any model element $e \in E_A \setminus S$.

  Using the same arguments made for the reversibility of this category, we do neither improve nor impede the theoretically inferenceable information on any $e \in E_A \setminus S$.

- **C4** The models $A$ and $B_{A,S}$ have to be behavioral equivalent.

  By only removing model elements that are covered by implicit assumptions we do not affect the behavior.
3.4 Limitations

The model transformation approach has its limitations. Even considering all categories presented, they can not obfuscate every model for every selection to not reveal any more than oracle access for unselected elements. We have examined the following three reasons to be the most serious:

- We can not prevent attackers from having a more detailed understanding of the domain. Additionally, having ignored laws so far, they could pose a problem as well (cf. Subsection 3.4.1).
- For some functions, it has been proven that it is impossible to obfuscate them into a form that reveals nothing more than oracle access (cf. Subsection 3.4.2).
- The formalization is very abstract. A concrete implementation may introduce further challenges. As an example, what is to be done, if the AST can not be transformed directly to a text-based formula (cf. Subsection 3.4.3).

Furthermore, we discuss general threats to validity in Subsection 3.4.4.

3.4.1 Possible Adversaries and Legal Issues

We can not control who tries to infer additional information from an obfuscated model. The adversary may be anything from an automated program to a domain expert with a deep understanding of the modeling domain. Furthermore, we cannot control the applicable law in all countries of the cooperating companies.

This results in one major limitation: While the behavioral equivalence can be checked and verified, the inference constraint is more difficult and very likely to be impossible to verify and even impossible to fulfill in general. This limitation is based on the following arguments:

1. Some functions have been proven to be impossible to obfuscate to reveal no information. They are discussed in Subsection 3.4.2.
2. Competition laws, which may restrict shareable information, are mostly informal and decided in precedents [18].

Different countries employ different competition laws, requiring different policies for each country. Furthermore, these policies may have to be updated with each reform to stay compliant [40, 47]. Moreover even humans and law experts have their difficulty to understand specific competition law restrictions. To quote from Making sense of competition law compliance:

- “Unlike other felonies, competition law violations are seldom obvious, and companies are not necessarily aware of their wrongdoing” [18, p. 8].
- “In addition to the legal uncertainties, the assessment of the legality of a certain conduct is often also dependent on factual circumstances which might be difficult to evaluate” [18, p. 8].

While some of the competition laws are relatively specific (e.g., it is forbidden to talk about or regulate prices), discussions on how to formalize other competition laws exceed the scope of this thesis, and it must be acknowledged that obfuscating models is only one part of staying compliant.

3. Adversaries may have domain knowledge using other sources to enrich the attack [28].

Even if it is possible to obfuscate the influence of a function, so it reveals only its input/output behavior, this may not be enough if the adversary can infer further information — for example, through the anomalies left by removed information. We discuss this further in Note 64/A below.
In general, it is hard to formalize the intuition and implicit knowledge an attacker might have of the model that is to be attacked. Consider a model about a car that describes in detail all compartments. For its obfuscation, only a small subset of elements is selected to remain, leading to an obfuscated model that misses components like the wheels, albeit their impact on the selected elements remains.

Even though the model does not contain wheels, the attacker might assume their existence based on its understanding of the concept “car”. Therefore it can use the “anomaly” of the indirect influence to make more knowledgeable guesses like “the requirements for the fuse that we are about to produce matches exactly those of the wheels manufacturer ExampleWheels”.

Dealing with such problems exceeds the scope of this thesis, and it may very well be impossible to deal with them completely. Therefore, we allow the modeling user to make manual changes to the automatically obfuscated model in the proof of concept implementation described in Chapter 4.

We could cope with at least some of those problems by introducing user-defined policies, essentially deferring the responsibility to a human authority. Those policies could be used to exclude information automatically [71, 37]. However, discussing these possibilities exceeds the scope of this thesis — we have restricted ourselves to limit user configuration to a minimum (i.e., in the context of tolerable behavioral equivalence).

### 3.4.2 Theoretical Limitations

There exist functions, that can not be obfuscated completely, albeit their effect on realistic models may be limited [64, 49, 58, 10].

As an example, we consider point functions. Point functions are boolean functions that yield 1 for exactly one input vector [85]. A prominent example of a point function is a password checker, which accepts exactly one password. While Narayanan and Shmatikov try to achieve better results for specific classes of point functions [64], Barak et al. prove that it is impossible to obfuscate any point function so that it does not reveal anything besides its input/output behavior [10]. Nevertheless, it may not be necessary to obfuscate functions completely to secure the IP embedded in them. In other words, it is up to time to show the practicability and legal implications of using “best-possible obfuscations” as described by Goldwasser et al. [50].

### 3.4.3 Limited Transformations and Formalization

While the transformation categories from Table 38/A were designed to cover a large range of applications, it is easily possible to devise a large range of special cases, where the categories presented do no longer suffice. A few examples:

- We have not discussed the possibility of functions written in another language that are embedded in the model. That is, a modeling language could allow the integration of functions and features written in another language that has to be obfuscated differently.

- When modeling in the context of big data, there is a lot more possible using the definition of tolerable behavioral equivalence. For example, it would be possible to create equivalence classes to anonymize age information — a technique common in the area of privacy-preserving data mining [81, 46].
The main reason for keeping the selection intact is that another modeling expert can still work with it. However, the recreated model only describing the model behavior may not be enough to allow working with the model efficiently or may even impede necessary analysis. This would require additional transformations that ensure usability.

We have not formally assessed the effectiveness of the transformation categories in terms of their resistance against attacks from automated and human users (cf. Subsection 5.2.2). However, the proof of concept implementation in Chapter 4 serves as a baseline for investigating this effectiveness.

3.4.4 Threats to Validity

Additionally to the limitations, our findings are subject to several threats to validity regarding the properties used and the generalization, similar to internal and external validity known from case studies [61, 86].

Missing Properties

Resembling the internal validity, which deals with the trustworthiness of the conclusions, the properties we have selected in Subsection 3.2.2 are subject to threats regarding their adequacy. While we have based our properties on other work focused on characterizing obfuscating transformations and model transformations in general [4, 3, 25], they may be insufficient in identifying all relevant aspects. Furthermore, the requirements we characterized for each category are not based on any prior research but stem from the intuitive understanding of the authors. Therefore, they may be not enough or even misleading when selecting transformations.

Generalization

Similar to the external validity — which deals with the generalization of applying the conclusions outside of their context — our approach is subject to problems when generalizing it to other modeling languages.

While we tried to produce a formalization that is as abstract as possible, we did not prove or show the generalizability of it, restricting the proof of concept implementation to IRIS. Therefore, further limitations in the application could stem from a limited applicability of the formalized approach.

Additionally, we did not focus on transformations that deal with the obfuscation of data as it is common in PPDM. This could result in further problems when applying additional transformations that deal with the anonymization of data (cf. Section 5.3).

3.5 Tracing and Execution Rules

This section briefly explains possible execution rules and highlights the importance of tracing to ensure termination and correctness [36, 76]. We use the term execution rule to collectively describe how and when to execute which transformation category from Table 38/A (cf. [29]). However, besides the execution rules presented, there are several other possible application strategies (e.g. [36]).
### 3.5 Execution Rules

#### 3.5.1 Subphases

The obfuscation phase of the program described at the beginning of this chapter is divided further into four subphases: i) outer, ii) inner, iii) structure, and iv) cleanup.

The subphases run sequentially, each being considered its own little program commanding over a certain subset of all categories employed. Subphases may use the same category differently — for example, by changing its rules of application.

#### 3.1 Outer

![Figure 66/A: Proposed scheduling for the four subphases in the obfuscation phase. Contains all transformations from Table 37/A. Each phase is to be read from left to right, transformations in the same block are executed in the same cycle.](image)

For a possible assignment of all categories to their individual subphase, see Figure 66/A. The figure will be used in the following segments.

#### 3.5.2 Outer Subphase

The goal of the outer obfuscation subphase is to remove as much information outside of the selection as possible. We propose the following scheduling:

![Diagram: Proposed scheduling for the four subphases in the obfuscation phase.](image)

Read from left to right the scheduling for the outer subphase presents the sequentially executed cycles A, B, C, D, and E, containing transformation categories.

Each cycle may be executed any number of times — specified by exhaustion rules. The categories assigned to each cycle are executed as presented from left to right for each iteration. If E is to be run two times, this results in the application: “T \(_9\) → T \(_2\) → T \(_8\) → T \(_9\) → T \(_2\) → T \(_8\)”. Additionally, for each category to be executed, the cycle may specify rules that select the normalized constraints or model elements to work on.

**Outer Subphase — Cycle A**

The first cycle runs only once and only executes Transformation Category 1 “Value Distortion”. By design, this category operates on each unselected variable that has been assigned a user configuration. Running it at the start ensures that we do no longer have to trace these distortions.

Because the category terminates, Cycle A terminates after one iteration.

**Outer Subphase — Cycle B**

Because reduce constants uses the closest bounds inferred by the solver, we can execute it prior to the evaluation categories. Executing it once it produces a finite number of new constraints.

Similar to Cycle A, through the termination of “reduce constants”, Cycle B terminates after one iteration.
Outer Subphase — Cycle C

The third cycle of the outer obfuscation subphases simplifies all calculations by performing constant propagation and constant folding operations. In order, each iteration causes the following transformations to execute:

- **T.7**: operation reordering ensures that all normalized constraints are in their normal form. By definition, this category operates on all selected and unselected constraints — it is executed once per iteration.

- **T.4**: arithmetic simplification evaluates functions. It is designed to accept a node of an AST that represents a constraint. Therefore, we execute it for each unselected normalized constraint in the normalized model.

- **T.5**: inline definitions operates, by definition, on each unselected normalized constraint. It is executed once per cycle.

- **T.10**: tailor expressions is called for each unselected normalized constraint in the normalized model.

In theory, the order in each iteration does not matter. If the arithmetic simplification makes a variable available for inline, it does not matter if the inline happens in the same or the next iteration of the cycle. No other category in this cycle can harm the applicability of another one. It can only aide the applicability other categories.

We consider this cycle to be “exhausted” if the tracing model records no new modifications after a complete iteration. By already having shown that each category terminates on its own, what remains is validating that their iterated sequential application does terminate as well.

Each expression has, by definition, exactly one normal form. Therefore, starting with the second iteration, T.7 modifies the normalized model only if another modification has occurred. Besides that, each inline removes a reference, and each arithmetic simplification removes at least one function call and add a finite number of references. The only problem is T.10 because, by definition, a tailoring function may exhibit any behavior. However, we can argue based on the tailoring intent. In other words: each function-call can only reasonably be tailored once, even if arithmetically simplified. If a function call can be tailored multiple times in sequence, we can argue that the initial tailoring could have exhibited the same behavior from the start.

Outer Subphase — Cycle D

Similar to the other single-category-cycles we can execute T.6 “Strength Manipulation” once for each unselected normalized constraint in the normalized model. Because the category terminates, Cycle D terminates as well.

Outer Subphase — Cycle E

All three categories of the final cycle serve a common purpose: raising obscurity. We can cycle any finite number of times, raising the obscurity arbitrarily.

3.5.3 Inner Subphase

The inner obfuscation subphase aims to dilute information outside the selection within the selected calculations. We propose the following scheduling (cf. Figure 66/A):

```plaintext
T.5  T.11  T.7  T.4
A    B    C    D
```
3.5 Execution Rules

Each cycle of this phase exists of only one category:

- **T.5**: inlines, by definition, all applicable unselected references into the selected normalized constraints. Hence it is executed only once.
- **T.11**: extends on T.5 by inferring closer known bounds for each selected reference. Just like inline, it is executed only once.
- **T.7**: is run once to ensure that all constraints are in a normal form again.
- **T.4**: dilutes the inlined bounds exactly as described in Note A.

All cycles are executed only once. Each category terminates for each execution and is only executed on a finite amount of elements. Therefore, the inner subphase terminates.

### 3.5.4 Structure Subphase

The goal of the structure obfuscation subphase is to recreate a model from the normalized model while breaking all remaining references to the original model and further obfuscate the unselected information. We propose the following scheduling (cf. Figure A):

![Scheduling Diagram]

Just like the inner subphase, the structure subphase consists of cycles with one transformation category each. Therefore, we review their application patterns:

- **T.12**: ensures, that no references from the original model remain. It is to be executed once for every constraint in the normalized model or alongside T.14.
- **T.13**: removes all normalized constraints that have no longer any impact on the behavior. It is executed once.
- **T.14**: proposes split and join which in term work against each other. However, we can use the tracing model again. Split is only used on nodes in the AST of constraints that either have not already been joined or that have been joined up to N times. Join may be called until there are M constraints left. This ensures termination.
- **T.15**: finally reconstructs the model and terminates with one iteration.

This subphase terminates. All cycles execute only one terminating category each. The cycles A, B, and D perform one iteration only, the cycle C terminates after a finite number of iterations. All categories are applied to a finite number of normalized constraints or model elements.

### 3.5.5 Cleanup Subphase

The goal of the cleanup obfuscation subphase is to raise the obscurity of the produced model and potentially remove redundant model elements. We propose the following scheduling (cf. Figure A):

![Scheduling Diagram]

The last subphase only executes two categories in two cycles:
3. One-Way Transformation

- T.16: similar to T.13, it is sufficient to execute the category once.
- T.3: may be executed N times with N > 1 if multiple layers of encryption are desired. In both cases, the cycle terminates after N iterations.

Cycle A is only executed once and terminates as T.16 itself terminates. The second cycle terminates after N iterations. Therefore, the cleanup subphase itself terminates.
Proof of Concept Implementation

We create the proof of concept implementation to reveal additional challenges in adopting the formal approach in practice and to show its applicability. While the implementation only covers a subset of all transformation categories from the previous chapter, it addresses a wide range of problems, like adapting the categories to a concrete language syntax and its semantics (i.e., IRIS). Elaborating on those “adoption”-challenges, this chapter discusses i) additionally implemented processes, and ii) the validation strategy used to verify the implementation.

4.1 Preparations

To allow an easy integration with the other features of the IRIS-project some preparation steps are required. While the “program” as described in Section 3.1 lays the groundwork, there are several features not offered or ignored by the formal specification. Furthermore, we can make use of preexisting parts of IRIS to achieve some of the transformation categories.

As this chapter is not intended to serve as an in-depth presentation of all features of IRIS, we focus the discussion on the following (cf. [15]):

1. **Concrete implicit assumptions** (cf. Subsection 4.1.1)
   All implicit constraints of the language have to be expressed explicitly. Otherwise, their implicit formulas may be wrong or not obfuscated enough. While this is part of the normalization phase, it is necessary to identify all implicit assumptions and derive a normalization for them.

2. **User Interaction** (cf. Section 4.3)
   The formalization expected the selection to be given. In a real-world scenario, the modeling user should be able to select the elements manually. Furthermore, we need to present the obfuscated model to allow for manual modifications and to automatically verify the constraints.

3. **Recovering lost information** (cf. Subsection 4.4.1)
   The original hierarchy of a model is destroyed by the flattening operations. Yet, it may be of interest to keep the structure of selected elements — for example, the ordering of the selected elements inside of a block — unchanged. Furthermore, IRIS already removes information when transforming a model into a normalized model. This requires additional work when restoring formulas into the domain-specific expression language in IRIS.

4. **Increasing usability** (cf. Subsection 4.4.3)
IRIS allows for human-readable aliases for generated unique identifiers. Furthermore, IRIS allows the modeling user to define value ranges and units for each property in separate graphical elements. In addition to that, it offers various additional elements like “Notes”, that have no influence on the behavior but may be kept in some cases to allow for a more obfuscated model which is more “usable”.

- What were the additional challenges of the implementation?
- How have these challenges been resolved?
- How was the implementation evaluated?
4.1 Preparations

Before discussing the two main tasks of the implementation: i) implementing the program from Chapter 3, and ii) designing the user interface, the remainder of this section covers the already existing parts of IRIS we use and the consequences their usage has.

IRIS is implemented in TypeScript\( ^{(2)} \) and uses the Preact\( ^{(3)} \) library which is an alternative to React. Tests are written with the Mocha\( ^{(4)} \) test framework.

### 4.1.1 The Solver of IRIS

While a more detailed look at the solving process of IRIS can be found in “A domain-specific language for modeling and analyzing solution spaces for technology roadmapping” [15, Section 5] we denote the most important points in which it differs from the “perfect” formal solver considered in Subsection 2.2.3.

Most importantly, the IRIS-solver allows relaxing the constraint system. Figure 72A on the left side illustrates this with a constraint \( C_1 \) that is relaxed to allow values \( X \in (0,1) \) as well. While this does not create any wrong result in the sense of too small bounds, it impedes the effectiveness of the obfuscation.

Consider a case in which Inclusion.\( P \) is used as the predicate in an unselected if-statement. The formal solver correctly infers Inclusion.\( P \) to be false at all times. However, the IRIS-solver does not — which affects the impact analysis.

Suppose, we select Obfuscate in Figure 72A and obfuscate the model. From the perspective of the IRIS-solver, applying \( C_1 \) does not affect the inferred results, as the bound for \( X \) remains the interval \([-3,3]\). Therefore, the process identifies \( C_1 \) as having no impact on the selected values and removes it. Figure 72B below illustrates the propagation of this relaxation to the obfuscated model. The figure contains the obfuscated model produced by the proof of concept implementation. The “Collector” block shown is explained in the next section.

In a nutshell, it contains elements that could not be obfuscated further.

In fact, Figure 72B illustrates even more problems of the current implementation. In theory, the “Collector” is completely redundant, as setting \( P = \text{maybe} \) would suffice. However, the proof of concept implementation currently lacks inlining operations that deal with “maybe”. This is due to the way the IRIS-solver itself handles uncertainties and is to be addressed in the future.

To improve the readability of the obfuscation examples, we have decided against random and large unique identifiers. Instead, we increment a counter — instead of generating a new random value — whenever a new identifier is requested. Furthermore, we do not choose the human-readable names randomly but include the unique identifier.

While this behavior is easy to change, it helps identify related fields in the obfuscated output on screenshots of the IRIS-frontend. For example, in Figure 72B the reference \([\#6] \) refers to \( P_6 \). The suffix 6 representing the unique identifier.

Because limitations regarding the relaxation of constraints are due to the solver implementation in IRIS and because these limitations affect the original model as well as the obfuscated variant, we ignore them for the implementation.
4. Proof of Concept Implementation

4.1.2 Specific Implicit Assumptions

IRIS has three implicit assumptions, two of which we have already discussed: the availability and the replacement assigned to blocks. The third one is based on the precise name of a property: “Gewicht” (translating to “weight”). If a property \( \rho \) is named “Gewicht” it automatically produces another constraint \( \rho \geq 0 \) kg enforcing this property to evaluate to a non-negative mass (cf. Figure 73/A).

We can express these assumptions explicitly. The availability and the replacement of a block can be overwritten by properties with the magic names “Availability” and “Replacement” respectively. For “Gewicht” we can produce another constraint element (cf. Subsection 2.1.2) stating the implicit constraint explicitly. However, whenever we use T.16 Remove Implicit, we have to take care not to remove a constraint that just coincidentally equals the implicitly created one.

4.2 Implementation Scope

We restrict the scope of the implementation through four rules. First, performance is of no concern as long as the time required for the obfuscation does not exceed a few minutes (cf. Subsection 4.5.3). Second, we only implement a subset of all presented categories to exemplify their application in IRIS. Third, the selection mechanism is limited to blocks, and fourth, we only support a subset of all features of IRIS. The first rule solely exists to clarify the focus of this implementation. We do not try to achieve a fast implementation but focus on an application of the formalization of Chapter 3 in practice. The following subsections explain and justify the other three rules in greater detail.

4.2.1 Adaption of Selected Transformations

Based on Table 38/A, the following transformation categories have been selected: T.4, T.5, T.7, T.11, T.12, T.13, T.15, and T.16 (cf. Subsection 4.2.1). Furthermore, T.10 has been implemented for some edge cases. In this section, we explain the reasons for choosing the individual categories alongside necessary adaptions.

First, IRIS offers no mechanism to rebuild a model from a set of constraints. Therefore, selecting the categories dealing with structural recreation and independence, T.12 Identifier Obfuscation and T.15 Rebuild Hierarchy is necessary to re-create an IRIS-model. Additionally, the existing IRIS-solver implementation does already provide transformations we can use to change the important categories T.4, T.5, and T.7. Due to their broad application range, we adapted them to work with the obfuscation program.

We implement T.10 Tailor Expressions and T.11 Reduce Constants rudimentarily for demonstration, yielding a deepened understanding of local time dependence and problems stemming from the conventional oberapproximation of the IRIS-solver (cf. [15]). We select T.13 Remove Orphaned because of its importance in removing information from the output model and as an example of impact analysis in IRIS. Finally, we choose T.16 Remove Implicit for convenience, reducing the size of obfuscated models in the examples.

Essential Adaptions

This segment explains the essential adaptations required to use the selected categories in IRIS. However, we do not explain the precise implementation for each category, nor do we explain every adaption necessary (e.g., translating the abstract definitions into TypeScript code), as they reveal no further challenges.
4.2 Implementation Scope

Most work went into the implementation of T.12 Identifier Obfuscation and T.15 Rebuild Hierarchy as both categories are the minimum requirement in order to ensure C/1 and C/2. For the implementation, we integrate T.12 into T.15 as suggested alongside their description. For each identifier requested by T.15 we query T.12 to provide it. Regarding the reconstruction of hierarchies, we deviate a little bit from the behavior described. This is due to two reasons:

1. **IRIS** requires all constraint and passive model elements to occur in a block. We address the first problem by producing a special Collector block that is addressed as the “upper most hierarchy” for unselected elements.

2. Constraint model elements with empty formulas do no longer exist in the normalized model if they are not referenced. To assure that all selected model elements remain in the obfuscated model, we use the original model (cf. Subsection 3.1.1) and recover all selected constraint model elements not present in the normalized model.

When restoring ASTs to the textual representation of the domain-specific expression language of IRIS, we employ some special cases which are described in Subsection 4.4.1 and further in Subsection 4.4.3. Furthermore, for each selected child block where the direct parent is unselected, we remove the parent relationship as shown in Figure 77/B with Block4.

To achieve T.4 Arithmetic Simplification and T.7 Operation Reordering we use the preexisting symbolic transformations the solver uses to infer bounds. Furthermore, we extended the inlining to allow restricting sources and targets according to the requirements of T.5 Inline Definitions. However, we have restricted the inlining to the unselected and do not inline bounds into selected constraints as the implementation effort exceeded the scope of this bachelor thesis (cf. Section 5.3).

Implementing T.10 Tailor Expressions is more difficult because IRIS offers no way of inferring the closest bounds for local time dependence. Therefore, we implement a basic approach that infers the closest domain for the local time dependence recursively as long as no other references are involved. However, we only implement the tailoring for special cases of the linear function as an example of how to implement tailoring functions. The application of the tailoring transformation category is shown in Figure 74/A illustrating a tailor to a constant value.

The implementation of T.11 Reduce Constants is simple, using the bounds inferred by the solver. However, determining the impacts for T.13 Remove Orphaned is rather hard for certain edge cases. We use the following approach. First, we build a dependency graph as described in Example 57/A and remove all constraints that do not have any impact. Afterwards, we use the “simple” approach of checking the impact of removing an unselected constraint using the dependency graph from before to remove all related constraints as well. To achieve T.16 Remove Implicit, we compare the normal forms of unselected constraints with those generated by IRIS and remove the unselected constraints if they are the same.
Regarding tracing, we only implement parts of the tracing model as described in Subsection 3.1.1, because IRIS does already record actions for each modification to a model. Therefore, we use this preexisting system to retrieve origin information and to trace all modifications.

4.2.2 Block-Based Selection Mechanism

We restrict the selection to a block-based mechanism that already exists in IRIS to reduce the implementation effort. However, this restriction only applies to the selection — the rest works independently from the selection mechanism.

For a block selected, we automatically select all properties, constraints, notes, and other elements directly assigned to it, but neither select children nor parents. Both implicit assumptions: the availability (cf. Note 11/A) and the replacement (cf. Note 11/B) are selected as well. In Figure 77/A the blocks “Block1” and “Block2” have been selected by the user which implies the selection of Block1.Gewicht, both constraints and implicit assumptions of Block1, Block4.Gewicht, Block4.R1 as well as all implicit assumptions of Block4.

4.2.3 Unsupported Elements

First and foremost, the main focus lies in creating an output model that suffices the four constraints. Therefore, we do not focus on recreating all passive model elements. While we do recover notes and discussions, there is no guarantee that we recover all of them. Furthermore, KPIs are recovered as properties to allow them to be referenced in other expressions — something IRIS does internally but does not allow in the expression language.

Besides those restrictions, we do not support some features that are special to IRIS because we focused on concepts like “blocks” and “properties” common in modeling languages. For example, IRIS does allow for runtime predictions which are evaluated by querying another server using configurations embedded in the model — we do not support runtime predictions in our proof of concept implementation. Additionally, we do not support a special element (named “Wirkung”) due to it being not fully implemented in IRIS itself. Finally, we restrict the set of supported exports inside a model.

Technically, IRIS allows the global time to be any point in time, using a floating-point number. However, the timeslider to set the current time works in steps of months and is always restricted by a finite interval. To verify the behavioral equivalence, we use the IRIS-solver to evaluate both models for each month in the time interval and compare the results for all selected elements.

4.3 User Interface

For the “selection” of model elements, we make use of the already existing selection mechanism (cf. Subsection 4.2.2). The proof of concept implementation includes three additions to the user interface of IRIS:

- **Obfuscate the current selection**
  
  As presented in Figure 77/A, we enrich IRIS by a new action “Export to obfuscated model” that initiates the obfuscation with the current selection.
4.3 User Interface

- **Present the obfuscated model to the user**
  After the obfuscation completes, the result is presented in another subwindow so the modeling user can verify the obfuscated result (cf. Figure 77/A). This window allows for any change to the obfuscated model.

- **Import the obfuscated model into another**
  Shown in Figure 77/A as well, we extend IRI5 to allow for the import of a model into an existing one with the action “Import JSON into Model”. It is explained further in Subsection 4.4.2.

The model preview is interesting because IRI5 did not allow for the presentation of multiple models. Figure 76/A illustrates the preview window, including the tooltip that is visible when hovering over the “صوم” status icon, which indicates a successful verification (cf. Subsection 3.1.4) of the exported model.

The timeslider at the top of the window in Figure 76/A — presented as a road with the car indicating the current time — allows setting a different time for the obfuscated model, which is of use when manual changes are desired. Changes are further assisted by the verification, which is re-run after each change of the model. This is shown in the Figure 76/B. The figures shows a change that breaks the behavioral equivalence by updating the maximum “Gewicht” in C18 from 10 kg to 12 kg. The violation is indicated by the “صوم” status icon alongside a highlighting of problematic elements and an explanatory tooltip.

We only highlight properties that directly violate the equivalence and not the elements that caused those violations indirectly because IRI5 already offers tools that explain the inferred bounds and influence of other elements.

### 4.3.1 Exporting the Obfuscated Model

Additionally, the preview window supports actions to i) synchronize the simulated time with that of the source model, ii) show or hide the timeslider, iii) replace the source model with the obfuscated variant, iv) import the obfuscated model into the source model, and v) download the model. They can be accessed via a click on the “ئ” menu icon as shown in Figure 78/A:
Figure 77/A: Example of obfuscating via the menu option “Export to obfuscated model” which is only enabled if at least one block is selected. The selection includes the blocks “Block1” and “Block4”. The preview of the obfuscated model is shown in Figure 77/B.

Figure 77/B: Preview of the obfuscated model (created in Figure 77/A) in a separate window.
4.4 Implementation Challenges

The proof of concept implementation reveals various challenges, of which the most relevant ones are discussed in this section. We start by addressing the problem of recovering information that is lost due to the flattening process performed by IRIS in Subsection 4.4.1. In Subsection 4.4.2 we discuss the import mechanism which allows importing an existing model into another one, followed by Subsection 4.4.3 that explains the steps to recreate elements that may aid a modeling expert working with the selected parts of the obfuscated model. At last, we explain the handling of solution alternatives and implicit assumptions briefly in Subsection 4.4.4 and Subsection 4.4.5.

4.4.1 Recovering Lost Information

The parser of the IRIS-solver already expands certain constructs. As an example, units are converted to a predefined base unit. Writing 5g results in a conversion to 0.005 kg with an annotation that the value should be displayed in gram. While this works perfectly for lower values, converting larger ones or converting between other units could result in rounding errors that harm the reconstruction of an expression translating to that value.

Furthermore, restoring identifiers is critical. As an example, IRIS does not allow to reference the implicit availability of a block by the “availability”-name. However, it allows using the block-name to access it. To cope with problems in recovering identifiers, we reconstruct identifiers by building the relative reference using the human-readable aliases in the form Parent.Block.A.X first. During the reconstruction we employ special cases that deal with the referencing of availability and replacements. After the reconstruction, we decide to use the unique identifier instead only in the case of a) naming collisions, or b) if an unselected element accesses a selected element or the other way around.

We illustrate an example of this reconstruction in Figure 76/A with the constraint C19: while the selected Gewicht is referenced using the human-readable alias, [6]#4 and [6]#8 are referenced by their unique identifier because they are unselected.

4.4.2 Importing Models

In the motivation, we proposed the use case of multiple parties collaborating with each other. To allow for this, we have to provide an import feature that allows loading an IRIS-model into an existing one.
However, when importing, we have to take care of the unique identifiers as we cannot ensure their global uniqueness. Therefore, we “obfuscate” all identifiers and update each expression of the imported model accordingly to ensure its unique identifiers do not conflict with any existing identifier. Furthermore, we prevent naming collisions whenever aliases are used as references by creating another parent block as a namespace containing all of the imported model’s blocks as children.

4.4.3 Increasing Usability

This subsection explains the various additional processes that we have considered to be beneficial to the usability of the obfuscated model. We start by explaining the model elements that we recover, describe how we populate additional fields in the properties, and discuss further possible cleanup operations.

Preserving Notes and Discussions

IRIS provides notes (cf. Subsection 2.1.2) that are of no importance to the behavior but may help a modeling user in understanding and using the model efficiently. We recover all notes of selected blocks by copying them from the original model.

Furthermore, IRIS supports a discussion feature that allows for a list of comments to be assigned to each model element. We copy the discussions of all selected model elements.

However, copying notes and discussions from the original model has the potential problem of introducing information of unselected information. Because both may contain arbitrary text, they can very well contain formulas or information worthy of protection. We have implemented this optional feature nonetheless to show its possibility and argue that introduced information can still be removed manually using the preview window (cf. Section 4.3).

Additional Types and Property Expressions

When recovering expressions alongside T.15 Rebuild Hierarchy, we trace the origins of selected constraints. If they originate solely from a model element expression, we do not create another IRIS-constraint but rather assign it to the formula of the selected model element in the output.

Furthermore, we recover type and unit information. IRIS allows specifying the desired type and unit of a variable using square brackets either in an expression (e.g., 5 [m]) or with an additional type field assigned to a property. We recover this information and update the expressions accordingly (e.g., by removing redundant type casts). This results in an intuitive recovery of selected elements, as illustrated by Figure 79/A on the right side.

Additional Cleanup Operations

Simply restoring expressions in the domain-specific language of IRIS based on their AST in the normalized model may result in very hard to read expressions. This is due to a lot of unnecessary type casts, parentheses, and long identifier names introduced by the IRIS-solver (e.g., by resolving all identifiers).

We deal with these problems during the generation by removing casts if they are unnecessary (i.e., they do not affect the type). Furthermore, we remove outer parentheses that do not affect the operation order and reconstruct identifiers as explained in Subsection 4.4.1.
4.4.4 Solution Alternatives

If all blocks that are part of an interface-implementation relationship are selected (cf. Subsection 2.1.1), we recover the relationship in the obfuscated model. Otherwise, we have decided against the recovery of partial relationships as this would require the introduction of placeholder implementations (or placeholder interfaces). We have decided this way because we can not stop IRIS from modifying the generated constraints in an interface-implementation relationship. Hence, to support the reconstruction of only partially selected interface-implementation relationships would require recreating all unselected replacement blocks.

Currently, we perform no further cleanup operations on the formulas in solution alternatives, resulting in large if-then-else constructs that are equivalent to those generated automatically by IRIS but that are hard to read and modify. Those additional cleanup operations remain future work (see Section 5.3).

4.4.5 Overwriting Implicit Assumptions

By default, we generate the availability and replacement overwrite for each block by producing properties with the magic names described in Subsection 2.1.4. While we do remove redundant overwrites afterwards, they remain whenever they have an impact on the inferred bounds, which is illustrated by Figure 80/A.

We choose this approach because it does not require placeholder blocks inside a block with unselected children. Hence, it allows us to obscure the number of children blocks. In Figure 80/A the resulting obfuscated model looks exactly the same if the requirement "Block B.R1" would be located in a child-block of Block B instead.

4.5 Evaluation

The proof of concept implementation’s primary purpose is to reveal further challenges when applying the formalization in practice. Therefore, we decide against the need for a case study assessing the usability in the given time frame of a bachelor thesis. However, we employ unit and system tests to assert the correctness of the implementation and conducted a performance benchmarking to gain an impression of the runtime behavior.

4.5.1 Study Design

We use proactive, regression-averse maintenance testing to ensure the proof of concept implementation produces correct results and is not affected negatively by further additions. For the benchmarks, we test models of different sizes with a common benchmarking library further described in Subsection 4.5.3. The...
models are selected from two sources:  
i) a collection of examples in IRIS, and
ii) a set of artificially crafted models that specifically target edge cases of the approach.

The collection of examples stems from multiple evaluations carried out in the context of IRIS (cf. [15]), each with participants of industrial partners. Hence, the examples are models — that resemble realistic scenarios — created to demonstrate the full feature set of IRIS, which makes them applicable for our evaluation as well.

The set of artificial models consists of models we have created to cover edge cases (e.g., colliding names or a deeply nested hierarchy).

## 4.5.2 Correctness

The core problem of proving the correctness is validating that the obfuscation program never harms the behavioral equivalence. Therefore, we use an implementation of the behavioral equivalences which checks the inferred results for each month in time for each selected property, availability, replacement, and requirement (cf. Subsection 2.1.2).

Based on this, we implement a set of unit tests that apply the obfuscation program to a collection of IRIS-models (cf. Subsection 4.5.1) and verify the behavioral equivalence. Furthermore, we employ a set of tests that verify edge cases and additional aspects — like the recovery of notes — that do not affect the behavioral equivalence. For each test, we run the obfuscation program for every possible selection (using the powerset of all blocks) and verify the behavioral equivalence individually.

Among the tested models, we include large variants (with over a hundred model elements each) that have been created by industry partners of the GENIAL!-project (cf. Subsection 4.5.1). Using the automated verification, we started with property-based testing by generating random models.

### Test Coverage

Using the 204 tests we have written specifically for the obfuscation by identifying the edge cases of each transformation category and using the models we mention in Subsection 4.5.1, we achieve a test coverage of 94.7\% of all statements and a branch coverage of 83\%. We can not achieve a test coverage of 100\% due to two reasons:

1. We use the “never” type in TypeScript making use of its typing system to catch missed cases at compile time. In other words, whenever we deal with a collection of different elements that have to be handled differently, we use never to assert that we deal with all of them. Furthermore, this assertion ensures that whenever another element is added to the collection, the programmer is notified of places that have to be adapted in order to deal with it. However, those statements can never be executed if all cases are covered, reducing the coverable code by a statement and a branch.

2. Neither do we test debugging functions that dump information during the obfuscation process nor do we test the construction of tooltips presented in Figure 76B — which leads to a low branch coverage, especially in the “Validation” group. Furthermore, due to the restricted implementation scope (cf. Section 4.2), some features are not tested but still covered in some functions to allow for the previously mentioned type checking at compile time.

---

Table 8/A: Test coverage of the proof of concept implementation showing covered statements (Stmts), branches (Branch) and functions (Funcs) according to MOCHA (cf. Note 72/A). The coverage is grouped by features. "Formulas" refers to the recovery of textual formulas from the IRIS constraints, "Categories" refers to the adapted transformation categories, and "Utility" refers to the tracing model and additional clean-up operations.
When running all obfuscation tests, the process takes about one hour. As this is certainly too long for everyday unit testing, we additionally implement a quicker version that requires only \( \approx 16 \) s. This quicker version uniformly picks random selections for large models and does not test different configurations but still achieves a total coverage of \( \approx 89.9 \% \).

### 4.5.3 Benchmarking

Although performance is not of great importance for the proof of concept implementation, we include some measurements of speed to give the reader a brief understanding of the efficiency of our implementation. We run all benchmarks using Node.js\(^{(9)}\) on Ubuntu 20.04.3 LTS using an Intel i9-9900K CPU with 16 cores and at least 16 GB of free RAM during the benchmark.

For each model described below, we benchmark every possible selection multiple times\(^{(9)}\) aggregating the mean time required. However, even models with a low number of blocks \( n \) result in a lot of possible selections (i.e., \( 2^n \)) and, therefore, a lot of measurements. Hence, some plots of the benchmark results — like that of the “Sensors” example in Figure 83/B with theoretically \( 2^{21} \) possible selections — only show a subset of all measurements uniformly chosen from all selection sizes.

For each plot of the benchmark results, each stacked bar represents an aggregated time measurement in milliseconds that is split between \( i \) the time required for the obfuscation, and \( ii \) the time required for the verification. The estimated overhead introduced by Node.js and the measurement itself is added in a light gray (albeit almost invisible most of the time because the overhead is negligible). Furthermore, error bars representing the margin of error due to the aggregation of obfuscation times are drawn as well.

The labels on the abscissa are to be understood as mere guidelines. From left to right, the abscissa represents an increase in model elements to obfuscate with the labels reading “obfuscated \( x \) elements due to the selection”. While the hidden labels are not to be interpolated linearly, we argue for that to be of lesser importance due to the following two reasons. First of all, because the precise time is of lesser interest, there is no added value in knowing “two elements required \( i \) ms more time”. Furthermore, the number of model elements selected reveals no information about the complexity of the expressions that are assigned to them and allows only to guess the complexity of the normalized model that is produced.

**EFuse Simplified**

The model “EFuse Simplified” represents a smart sensing fuse and is used by Breckel et al. as the running example \([15]\). It consists of 9 blocks and 34 model elements — including implicit assumptions — that are of interest for the obfuscation. Furthermore, it contains children and interface-implementation relationships (cf. Subsection 2.1.1). The time is restricted to lie within the years 2019 and 2040.

Benchmarking results are displayed in Figure 83/A and show that the majority of time is spent on verifying that the result is correct (with a maximum time of \( \approx 397 \) ms). This is due to the time restriction, which yields 253 points to verify when using a resolution of months. All 253 points have to be checked for each selection and for each benchmarking run by solving the complete model and comparing all values in common. Thereby, the impact on performance is mainly due to the solver’s performance integrated into \( \text{IR1S} \) and beyond the scope of this thesis.
The time required for the verification in Figure 83/A varies widely between different selections (ranging from 3 to 34 model elements). It averages to $\approx 199$ ms with a standard deviation of $\approx 84.6$ ms. This variation is due to the different model elements selected. Inferring a value for $b$ with a constraint $b = 3$ is much simpler than solving multiple interdependent constraints each time.

**Sensors**

The “Sensors” model was created by experts from the industry with realism in mind. It consists of 21 blocks and 165 model elements that are of interest for the obfuscation. In addition, it contains an interface with fourteen viable implementations. Due to the size of $2^{21}$ possible selections, we have restricted the benchmarking shown in Figure 83/B to roughly 540 selections uniformly selected from all sizes. The external factor time is restricted to lie within the years 2019 and 2036.

The average obfuscation time is 219.23 ms, with a standard deviation of 211.95 ms, while the average verification time is 205.73 ms, with a standard deviation of 102.37 ms. Especially the large standard deviations show that this benchmark is different. They are due to the aforementioned block with fourteen different implementations — the time required to obfuscate and verify the model is governed by the fact if this block is selected (drastically reducing the information to obfuscate while increasing the information verify) or not. Combined with the fact that we have chosen the displayed selections uniformly from all sizes, this explains why the growing size of selected model elements raises the chance for a shorter obfuscation time: the higher the number of selected elements, the higher the chance the specific block is selected. The time required for the verification is relatively low because large parts of the model are not interdependent, allowing the solver to assign the closest bounds directly.

Most time during the obfuscation of the “Sensors” model is spent on the impact analysis of constraints (cf. Subsection 4.2.1) in I/R.sc/I.sc/S.sc. This is due to its naive implementation that focuses on being correct. However, the naive implementation can be improved in the future.
4.5 Evaluation

Other Models

Additionally to the two previously presented models “EFuse Simplified” and “Sensors”, we have benchmarked all other models (cf. Subsection 4.5.1). Subsection 4.5.3 illustrates the obfuscation result of the example from Figure 10/A. Similarly, Figure 84/B shows the result of a randomly generated very large model with over 300 model elements to obfuscate.

![Obfuscation Verification graph]

We decided against the inclusion of all benchmarks in this thesis as they produce similar results and are not interesting for further analysis. However, they are included in the digital attachments (cf. Appendix C). All performance benchmarks yield an obfuscation time far below one minute, satisfying the performance restriction from Section 4.2. However, there is room for improvement. For example, by optimizing the impact analysis from Subsection 4.2.1 as mentioned in Note 83/B.

![Obfuscation Verification graph]

4.5.4 Threats to validity

Similar to case studies, our approach is subject to several threats regarding its reliability as well as its internal and external validity [86]. We discuss them in the following segments.

Reliability and Reproducibility

While we can not legally share all models used, all scripts used for the benchmarking are part of the digital attachments listed in Appendix C.4. Namely, the following two files are essential for the benchmark: benchmark_obfuscations.ts and benchmark-parse.

Internal Validity

The internal validity of a study describes to what degree of confidence the causal relationships between i) the tests and our argument for the correctness of the implementation, and ii) the benchmarks and our assessment of the implementation performance are not influenced by other factors.
For the testing strategy, the testing coverage of 94.7% could result in uncovered cases that may lead to incorrectly obfuscated models. Furthermore, limitations of the IRIS-Prototype (cf. Subsection 4.1.1 and Section 5.2) are neither tested nor treated specifically. Hence, if the IRIS-solver infers wrong results for a given model, the verification of the behavioral equivalence is wrong as well. However, we counteract these problems by adding manual test cases that check for required elements in the output model and that specifically test for models that are not behavioral equivalent.

Our benchmarks are subject to two major threats to internal validity. First, we are using IRIS which itself is a scientific prototype. Hence, IRIS is — similarly to our proof of concept implementation — not optimized for performance. Second, other processes running on the same machine during the benchmarks may have influenced the required time. While we have tried to keep the number of other processes running to a minimum and repeated the benchmark three times (with similar results), we cannot ensure that they had no effect on the benchmark result. Nevertheless, we consider the results to be sufficient. Because, although we cannot rule out other factors, all of those factors have a potentially negative influence on the required time at most. Therefore, we are satisfied with the maximum obfuscation times measured by the benchmark being that low — even low enough to not interrupt the user's flow of thought according to Nielsen [65, p. 135] — albeit neither the IRIS-prototype nor our proof of concept implementation is optimized for performance in any way.

Tests and benchmarks were subject to one problem of Node.js. When trying every possible selection for very large models (i.e., more than 20 blocks), we encounter problems with a limitation of the heap memory. We counteracted this problem in two ways: i) we choose only a subset of all selections uniformly from all sizes, and ii) we run the tests multiple times in sequence — always using another subset of all selections — to test all possible selections.

**External Validity**

The external validity of a study describes the generalizability and transferability of the presented findings to other models and modeling environments.

The benchmarking and the tests written are specific to the proof of concept implementation in IRIS and the adaptations described in Subsection 4.2.1. While the tests are independent of individual models — testing primarily the specific cases occurring inside those models — comparing benchmarks like that of “EFuse Simplified” (cf. Figure 83/A) and “Sensors” (cf. Figure 83/B), reveals a heavy dependence on the individual model to be obfuscated. We counteract this by using various models created by the industry partners of IRIS with realism in mind and combine them with artificially created models that target the edge cases of specific categories. Furthermore, we focus our proof of concept implementation on concepts (e.g. “blocks” and “properties”) that are common in other modeling languages.

Nevertheless, none of the models in the test set is a real model used in the industry. If those models are significantly larger, the verification and obfuscation time might not scale very well. However, we argue for this problem to be of lesser impact because even though the prototype implementation of IRIS itself has problems displaying significantly larger models, it is perceived well by the domain experts during the evaluation [15].
5 Conclusion and Future Work

As an overall summary, we use this chapter to describe general problems encountered during the writing of this thesis and reiterate the most important points. Furthermore, we discuss possible future work in the area of one-way transformation.

5.1 Conclusion

Collaboration between multiple partners is crucial for modern businesses. In this bachelor thesis, we propose i) a formalized automated one-way model transformation approach to support a compliant model exchange between businesses while allowing for a selection of elements inside these models to remain, and ii) a proof of concept implementation in the graphical modeling language and tool IRIS.

Section 1.2 formulates the problem statement and reduces the problem to four major constraints that the model transformation approach has to fulfill in order to produce what we call a “huskied model $B_{A,S}$” based on a given model $A$ and a selection $S$: i) $B_{A,S}$ has to be self contained, ii) $B_{A,S}$ has all elements from $S$ but none from $E_A \setminus S$, iii) $B_{A,S}$ allows no inference on any element $e \in E_A \setminus S$, and iv) $B_{A,S}$ and $A$ have to be behavioral equivalent.

Based on these four constraints, we derive three requirements for the implementation: i) the implementation has to satisfy all four constraints, ii) the huskied model has to be an ordinary IRIS-model, and iii) for each obfuscation, IRIS has to present a preview of the huskied model to the user. Along with the research questions from Section 1.3 and the research method described in Section 1.4, Chapter 3 proposes a program that fulfills all four constraints. This program consist of four phases: i) the normalization, ii) the preparation, iii) the obfuscation, and iv) the verification phase. The focus of this thesis lies on the third phase — the obfuscation phase — which is divided further into four subphases. Using the formalization from Section 2.2 (answering RQ4), Table 3B/A proposes sixteen categories — like $T_{12}$ Identifier Obfuscation and $T_{15}$ Rebuild Hierarchy — for the one-way model transformation to be performed in the obfuscation phase. We define and analyze seven major properties for all sixteen transformation categories. Additionally, we discuss the transformation categories regarding their influence on the four major constraints and propose execution rules that guide their application in Section 3.5.

Chapter 4 illustrates the second artifact — the proof of concept implementation in IRIS which is first explained in Section 2.1. We explain necessary adaptions and restrictions when applying the program of the previous chapter in practice. Afterwards, Section 4.3 describes the most important additions to the user interface and Section 4.4 covers the most important challenges of the implementation. Finally, Section 4.5 deals with the testing strategy and benchmarks the approach. We explain our proactive, regression averse, and maintenance testing strategy.

• How applicable is the solution?
• What are its shortcomings?
achieved a test coverage of \( \approx 94.7\% \) with a total of 204 tests. Furthermore, our benchmarks — using synthetic models and models designed by the industry partners of IRIS — show that our implementation is reasonably fast and does not interrupt the user’s flow of thought (cf. Nielsen, [65, p. 135]).

Regarding RQ1, we cover a wide variety of preexisting tools like ANONYMOUSXL and MONDO in Section 2.4. Furthermore, we argue, that none of them satisfies all four constraints. In the context of RQ2, we identify competition laws and the three requirements for the proof of concept implementation as specific requirements for the target model. With Chapter 3 we propose a set of transformations to answer RQ3, dealing with RQ3.ii) in Section 3.2 and exemplifying their flexibility with the proof of concept implementation. Alongside, we describe behavioral equivalence — a mechanism to compare the input output behavior of two different models.

In the course of this thesis, we add a new perspective to the existing solutions supporting compliant collaboration in the context of collaborative modeling by proposing our i) formalized and automated one-way model transformation approach, and ii) the proof of concept implementation in IRIS that demonstrates the applicability of our formal solution. Furthermore, we enrich IRIS with new mechanisms for obfuscating models and promote further research on the potentials and drawbacks of one-way model transformations.

### 5.2 Problems and Limitations

Section 3.4 and Section 4.4 already covered a large part of the challenges encountered. This section covers i) problems caused by the implementation-specific restrictions of IRIS, and ii) problems with the overall approach.

#### 5.2.1 Restrictions of IRIS

IRIS contains various implementation-specific restrictions. However, the most critical one encountered is a current implementation bug that heavily impedes the scope of models that can be obfuscated correctly.

---

**Figure 88a:** Example of the IRIS-solver inferring wrong bounds. In the Example block, All is inferred to \([14150 \text{km} .. \infty \text{km}]\). However, when extracting the right part of the intersect in Information, P3, the bounds change to \([22650 \text{km} .. 23850 \text{km}]\), albeit extraction should not change the results. This is shown with block Example2.

---

**Figure 88a:** Example of the IRIS-solver inferring wrong bounds. In the Example block, All is inferred to \([14150 \text{km} .. \infty \text{km}]\). However, when extracting the right part of the intersect in Information, P3, the bounds change to \([22650 \text{km} .. 23850 \text{km}]\), albeit extraction should not change the results. This is shown with block Example2.

---

**Example:**

\[
\text{All: } \text{Information.P1} + \text{Information.P2} + \text{Information.P3} \\
\text{Ci: } \text{Information.P1} \geq 0 \text{km}
\]

**Example 2**

\[
\text{All: } \text{Information.P1} + \text{Information.P2} + \text{Information.P3} \\
\text{Ci: } \text{Information.P1} \geq 0 \text{km}
\]

---

**Information:**

\[
P1: \text{P2(T) + 8500 km} \in [0 \text{km} .. \infty \text{km}] \\
P2: [0 .. 1200] [\text{km}] \in [-8500 \text{km} .. 0 \text{km}] \\
P3: [0 .. \infty] [\text{km}] \cap ([-1] \cap 14150 \text{km})
\]

---

**Information:**

\[
P1: \text{P2(T) + 8500 km} \in [0 \text{km} .. \infty \text{km}] \\
P2: [0 .. 1200] [\text{km}] \in [-8500 \text{km} .. 0 \text{km}] \\
P3: [0 .. \infty] [\text{km}] \cap ([-1] \cap 14150 \text{km}) \\
X: ([13900 .. \infty]) [\text{km}] \cap 14150 \text{km}
\]

---

Figure 88a above showcases this bug. While the specific formulas are of no importance, Example and Example 2 do only differ in that Example 2 has extracted the right part of the intersection in Information.P3 to Information.X. Interestingly, the bounds inferred for Information.P3 to Information.X. remain the same, while the bounds for the three properties All, Information.P1, and Information.P2 differ. In other words, extracting X “fixes” the values inferred. Because the IRIS-solver often generates a lot of intersections during the solving process, this results in a faulty impact analysis which can lead to errors in the obfuscated model.

---

[65] Jakob Nielsen
Usability engineering.

Section 2.4 on page 22.

Section 3.2 on page 34.

Behavioral Equivalence: All variables in common evaluate to the same value for all external factors.
We were unable to fix this bug during the writing of this thesis. This leads to the problem of manually tracing the source of a failing test each and every time. A lot of errors revealed during testing ultimately stem from the bug mentioned above. Moreover, this impedes the examples that can be used for testing and benchmarking — although we argue for this to be a lesser problem due to the high test coverage discussed in Subsection 4.5.2 achieved without the usage of models affected by this bug.

5.2.2 Limitations regarding the Obfuscation

In the time frame of a bachelor thesis, we were unable to conduct a study regarding the effectiveness of the model transformation approach when attacked by automated or human users [83]. However, this does not affect the arguments in Chapter 3 but only the effectiveness of the approach as a whole (e.g., identifying the need for new transformations for certain cases). Such a study can be carried out as future work and is covered in the next section.

5.3 Future Work

There are a lot of potential topics for future work that can provide additional insights into the efficiency of the presented approach and its accompanying implementation.

First of all, it would be interesting to implement the other transformation categories presented in Chapter 3 and assess their efficiency in a case study as mentioned in Subsection 5.2.2. This case study could reveal the need for further transformations that prevent the exhibited attack patterns. Furthermore, a case study with automated and human users can expose potential weaknesses of the approach.

Furthermore, another case study can assess the usability of the user interface (cf. Section 4.3). An adequate number of participants can provide further insights into how to present obfuscated models to the user, improve the editing and obfuscation experience, and help present obfuscated elements when importing an obfuscated model.

Additionally, these studies can be coupled with the integration of data obfuscations known from privacy-preserving data mining [46], for example, by analyzing the impact of obfuscating name and IP address information in a model. Furthermore, they can assess the effects of more noise introduced into the model [77], for example, by splitting the “Collector” block and introducing artificial relationships between blocks.

Besides the studies mentioned, it would be interesting to explore the capabilities of hybrid approaches. That is, combining the one-way model transformation from this thesis with other obfuscation approaches like the model fragmentation proposed by Goettelmann et al. [48]. The fragmentation could be used in edge cases to ensure critical information is never distributed while still allowing the receiver to perform offline with large parts of the model.

Moreover, interviews with modeling domain experts from the industry would be a good starting point to employ further clean-up operations on the selected elements. For example, it could be beneficial to extract all information from unselected elements to the Collector and remove them from the selection, so the modeling experts do not have to deal with them. This extraction would require additional transformations to perform that task.


This chapter may be used as a reference during reading and as a starting point for further research. It contains all sources and lists of the most important terms, acronyms, and notations used in this thesis.

Links that occurred for further information only are not listed again. All graphics have been created by the authors with TikZ\(^{(1)}\) if not stated explicitly otherwise. All \textsc{iris}-screenshots were taken by an automated process controlled through the \LaTeX-sources and written by the authors as well. They are based on the latest development version at submission-time.

\section*{A.1 Bibliography}

\begin{itemize}
  \item List of all references.
  \item Lists on used terms, acronyms, and notations.
\end{itemize}

\footnotetext[1]{https://www.ctan.org/pkg/pgf (10/11/20)}

\footnotesize

\begin{itemize}
\item \cite{akehurst2005kent}: David H. Akehurst, W. Gareth Howells, and Klaus D. McDonald-Maier. “Kent model transformation language”. In: \textit{Model Transformations in Practice Workshop, part of MoDELS}. 2005 (see p. 2).
\item \cite{amrani2012tridimensional}: Moussa Amrani et al. “A Tridimensional Approach for Studying the Formal Verification of Model Transformations”. In: \textit{2012 IEEE Fifth International Conference on Software Testing, Verification and Validation}. 2012, pp. 921–928. \textsc{doi}: 10.1109/ICST.2012.197 (see pp. 35, 65).
\end{itemize}

---
A1 Bibliography


A. Sources


Behavioral equivalence  Two models A and B have the same observable behavior for all variable they do have in common. It is described in Subsection 2.2.3.  
(see pp. 2, 4, 6, 17, 20, 23, 33, 35, 40, 42, 44, 46, 47, 49, 51, 53, 55, 63, 68, 75, 76, 79, 81, 85, 87, 88)

Constraint model element  Component of a model and specific type of a model element. These model elements contain an expression that is either taken at it is or transformed by a determining, finite process that depends on the specific type of constraint model element.  
(see pp. 16, 18, 59, 61, 74, 97)

Execution rule  Describes the application, scheduling, and exhaustion of transformation categories.  
(see p. 65)

External factor  A factor defined outside of the model and its expressions. Its change can yield completely different values inferred by the solver.  
(see pp. 4, 19, 21, 31, 35, 38, 40, 42, 45, 46, 49, 53, 56, 83)

Huskied Model  One-way transformed model satisfying the four constraints from Section 1.2.  
(see pp. 5, 25, 56)

Model  Finite set of model elements. As an example, the model $A = \{a : 3, b : 5\}$ consists of two model element: a and b. The set of all model elements is denoted as $E_A = \{a, b\}$.  
(see pp. 2, 5, 16, 21, 28, 47, 49, 64, 66, 68, 71, 73, 75, 83, 87, 97, 98, 100)

Model element  Component of a model. It consists of a unique identifier and further information that depends on the concrete type. The three types of structural, constraint, and passive model elements are defined further in Section 2.2.  
(see pp. 2, 5, 16, 21, 28, 44, 46, 47, 49, 51, 53, 55, 64, 66, 68, 71, 73, 75, 79, 87, 97, 98, 100, 102)

Normalized constraint  An expression that follows the predefined syntax rules that are understood by the transformation program and has to be satisfied by the solver. Referred to as “constraint” or “fact” as well.  
(see pp. 18, 19, 28, 29, 31, 33, 35, 43, 52, 54, 62, 66, 68, 71, 73, 74, 79, 82, 83, 97)

Normalized model  Flattened finite set of normalized constraints mapped to their origin model elements. All implicit assumptions have to be expressed explicitly. Furthermore, joined constraints have to be split as described in Section 2.2.  
(see pp. 18, 28, 29, 31, 33, 44, 47, 50, 56, 62, 67, 68, 71, 74, 79, 82, 100)
**P**

Passive model element  Component of a model and specific type of a model element. Passive model elements have no direct effect on the model behavior. They can serve as mere comments, or they can be used to name certain relationships by functioning as an alias. However, they do not generate any constraint, so removing them is not allowed to affect the inferred value for any element, as long as the effects or potential aliases are resolved. *(see pp. 16, 18, 50, 74, 75, 97)*

**S**

Structural model element  Component of a model and specific type of a model element. Like the name suggests, structural model elements define the hierarchy of a model. A structural model element may contain any number of other model elements recursively and may imbue them with additional semantics. *(see pp. 16, 18, 28, 60, 61, 97)*

**T**

Tolerable behavioral equivalence  Two models A and B have the same observable behavior for all variable they do have in common, based on a comparator function $\text{comp}_e(x, y)$ that accounts for thresholds: $\forall t \in [t_s, t_e] \forall e \in V^E_A \cap_m V^E_B : \text{comp}_e(e_A(t), e_B(t))$. *(see pp. 20, 35, 39, 41, 64)*

Transformation category  Groups arbitrarily many single transformations that do serve the same purpose. A transformation category represents an elementary operation, albeit consisting of multiple transformations that may deal with specific cases. *(see pp. 19, 30, 69, 71, 73, 74, 81, 82, 85, 87, 89, 97)*

**V**

Variable  As only a subset of all model elements has an effect on the solver or may be assigned values, we use the term “variable” to refer to arbitrary model elements that receive a value by the solver. *(see pp. 2, 4, 6, 17, 20, 23, 33, 35, 39, 42, 44, 46, 47, 49, 51, 53, 55, 64, 66, 68, 75, 76, 79, 81, 85, 87, 88, 97, 98, 100, 102)*

**Acronyms**

**A**

AST  Abstract Syntax Tree  *(see pp. 19, 32, 42, 44, 46, 49, 52, 54, 57, 59, 60, 63, 67, 68, 74, 79, 101, 105)*

**B**

BMBF  German Federal Ministry of Education and Research. From german: “Bundesministerium für Bildung und Forschung” *(see p. iii)*

BPM  Business Process Model  *(see pp. 23, 105)*

**C**

CAD  Computer-Aided Design  *(see p. 23)*
A. Sources

**DFG** German Research Foundation. From german: “Deutsche Forschungsgemeinschaft”  (see p. iii)

**FMI** Functional Mockup Interface  (see p. 26)

**IP** Intellectual Property  (see pp. 1, 2, 22, 23, 48, 64)

**IP address** Internet Protocol address  (see pp. 5, 23, 24, 29, 89)

**IRIS** (IrIS) Interactive Roadmapping of Innovative Systems  (see pp. 2, 8, 9, 105)

**KPI** Key Performance Indicator  (see pp. 9, 12, 18)

**MDA** Model-Driven Architectures  (see pp. 2, 21)

**NAT** Network Address Translator  (see p. 25)

**OWL** Web Ontology Language  (see p. 24)

**PIM** Platform Independent Model  (see p. 21)

**PPDM** Privacy-Preserving Data Mining  (see pp. 5, 23, 24, 65)

**PSM** Platform Specific Model  (see p. 21)

**QoS** Quality of Service  (see p. 23)

**SI** The Internationl System of Units. From french: “Système international d’unités.”  (see p. 3)

**UML** Unified Modeling Language  (see pp. 2, 9, 105)
Huskied Model \((B_{A,S})\) A model \(B\) created from another model \(A\) with selection \(S\) by the process described in this thesis. \(\text{ (see pp. 5, 33, 39, 41, 43, 44, 46, 47, 49, 51, 53, 55, 57, 58, 61, 62, 87)\)

Integer \((Z)\) Describes the set of non negative and negative integers. \(\text{ (see p. 52)}\)

Model-Elements \((E_M)\) Set of unique identifiers of the model elements in model \(M\) or the set of the unique identifiers of the model elements denoted as origins in a normalized model \(M\). \(\text{ (see pp. 3, 4, 18, 100)}\)

Natural Number \((N)\) Describes the set of non negative integers. If the 0 is to be included, it is indexed as \(N_0\). \(\text{ (see p. 51)}\)

Powerset \((P)\) With \(P(X)\) we denote the powerset of a set \(X\). That is, the set of all subsets of \(X\), including the empty set and \(X\) itself. \(\text{ (see p. 32)}\)

Real number \((R)\) Describes the set of real numbers. \(\text{ (see p. 44)}\)

Variable Model-Elements \((V_{EM})\) Filters from \(E_M\) all the model elements that are variables as described in Section 2.2. \(\text{ (see pp. 17, 20, 57)}\)
This appendix lists all rules and functions that are used in the pseudocode notation of the transformation categories.

B.1 General Rules

In the various pseudocode descriptions of the transformations, the used functions are defined as follows. For each definition, all names encapsulated in angles (e.g., ⟨example⟩) refer to parameters that can be passed to those functions. Everything that is written in bold (e.g., children) is a reserved keyword that has to be written precisely in this way. The /-Symbol is used to separate possible alternative styles. Explicit types are omitted and defined by the describing text.

Please note, that logical (and, or, xor, ...) and mathematical operators (+, −, *, ...) are available with their intuitive definition. Furthermore, predicates (denoted with a ★) with the same parameter list may be joined together. As an example, the expression M, N, 0 are known constant leaf evaluates to true if M, N, and 0 fulfill the joint-predicate known constant leaf. That is, they have all to fulfill known, constant and leaf individually.\(^{(1)}\)

For assignments we use the a ← b notation, to be read as “a replaced by b” [55]. break is assumed to exit the innermost loop. Likewise, return is assumed to exit the innermost function.

All functions are assumed to terminate and to determine if not stated differently.

B.2 Reference

B.2.1 Tree Related

node(⟨⟩) / node(⟨v⟩) / node(⟨v⟩, ⟨c⟩, ...) Returns a new node that may have the arbitrary value v. If no value is given, the created node is considered empty. If c is given, it represents an arbitrary amount of children which are assigned to this node. If those children already had a parent node, they are re-parented.

children(⟨n⟩) Returns a tuple of all direct children from a node n in a tree-like data structure (e.g., an AST). This function is deterministic: it always returns the children in the same order.

subtree(⟨n⟩) Instead of children, this returns the subtree, with n as root node. If n is a leaf, this returns just n.

leaf(⟨n⟩) ★ A predicate, checking if the given node n in a tree-like data structure is a leaf (i.e., children(n) returns the empty set).
leaves((a), (b), ...) / leaf((a)) and leaf((b)) and ...

★ Same as leaf, but for an arbitrary amount of nodes as parameters.

B.2.2 Modifying Data Structures

remove ⟨e⟩, ... from ⟨d⟩

Removes e from the data structure d. Can be viewed as a replace with n being nothing.

replace ⟨e⟩, ... from ⟨d⟩ with ⟨n⟩, ...

Replaces all elements e in the data structure d with elements n. There are two valid cases:

1. There is exactly one e and one n given: e is replaced directly with n.
2. There are multiple e and the same amount of n given: Every e is replaced by the n of the same index.

This expression is defined only for those two cases and no other.

B.2.3 Functions

signature(⟨s⟩)

Consumes s to build a predicate that accepts the value of a function as an argument and checks if f provides an implementation for the signature s.

Therefore, it is possible to write f has signature(a, b, c). An expression that evaluates to true if f offers an implementation with the given signature a, b, c.

function(⟨f⟩) / function(⟨f⟩, ⟨s⟩)

★ This predicate returns true if f is a function. If s is given it represents the expected signature and therefore enforces the predicate to return true if f is a function and if f is available with the signature s.

functions(⟨a⟩, ⟨b⟩, ...) / function(⟨a⟩) and ...

★ This predicate works similar to function but allows multiple parameters to be given. It returns true if all of the parameters satisfy function individually.

B.2.4 Values

constant(⟨v⟩)

★ This predicate checks whether v is constant. We define v as a constant if and only if it is no reference, no external factor, no interval, and no error. For example, 5, true, and −π are constant, but x and [1, 5] are not.

finite(⟨v⟩)

★ This predicate checks if v is finite. For v to be finite it has to be a constant representing a number and this number has to be finite (i.e., not ±∞).

value(⟨e⟩)

Returns the value assigned to an element. For a wrapper (like a node), this is the wrapped data. For a function, this returns information that can be analyzed further by the function predicates.

reference(⟨r⟩)

★ This predicate returns true if r is a reference.

domain(⟨v⟩)

Returns the domain of a variable-reference that is inferred by the solver (as described in Subsection 2.2.3).
B.2.5 Utility

\( \langle x \rangle \text{ is } \langle p \rangle \) / \( \langle p \rangle(\langle x \rangle) \)
Returns true exactly if \( x \) fulfills the predicate \( p \) and false otherwise. Therefore it is another way of writing the application: \(<p>(\langle x \rangle)\) to stay consistent with \textit{are}.

\( \langle a, b, \ldots \rangle \text{ are } \langle p \rangle \) / \( \langle a \rangle \text{ is } \langle p \rangle \text{ and } \langle b \rangle \text{ is } \langle p \rangle \text{ and} \ldots \)
Returns true if and only if all \( a, b, \ldots \) fulfill the predicate \( p \) and false otherwise.

\( \langle x \rangle \text{ has } \langle p \rangle \) / \( \langle p \rangle(\text{value}(\langle x \rangle)) \)
Returns true if the \textit{value} assigned to \( x \) fulfills the predicate and false otherwise.

\( \langle a, b, \ldots \rangle \text{ have } \langle p \rangle \) / \( \langle a \rangle \text{ has } \langle p \rangle \text{ and } \langle b \rangle \text{ has } \langle p \rangle \text{ and} \ldots \)
Just like \textit{has} but for multiple values.

\( \langle x \rangle \text{ has } \langle p \rangle \text{ as } \langle n \rangle \)
Works just like \textit{has}, but if \( x \) really fulfills the predicate according to \textit{has}, the \textit{value} of \( x \) is assigned to the name \( n \).

\( \langle a \rangle \text{ equals } \langle b \rangle \)
A comparison just like \( a = b \) without any implicit conversions or unwrapping. This means, two references are equal if they point to the \textit{identical} thing.

\textbf{not} \( \langle p \rangle \)
\[ \star \] Invert a given predicate \( p \), so that the new predicate returns true if and only if \( p \) returns false and true otherwise.

\textbf{known}(\langle x \rangle)
\[ \star \] We know \( x \) if it is present in its according database. In other words: if \( x \) is a \textit{function} it has to be a known function (cf. Subsection 2.2.2), if it is a \textit{reference}, it has to be a known reference (cf. Subsection 2.2.2), ...

\textbf{random}(\langle s \rangle)
Returns a truly random value from a set \( s \). If \( s \) contains no element, this function is not defined. This function does not determine.

\textbf{filter} \( \langle a \rangle \) \( \langle p \rangle \)
Returns from the collection \( a \) exactly the elements that fulfill the predicate \( p \).
Further Lists

List of Figures, 105  •  List of Tables, 106  •  List of Examples, 106  •  List of Digital Attachments, 106

This final appendix contains lists of all figures, tables, and examples that have been used in this thesis for quick reference. Furthermore, it contains a list of all digital attachments.

C.1 List of Figures

Figures repeated for referencing do not occur multiple times in this list.

3/A  A simple model ........................................ 3
4/A  Model transformation sketch .......................... 4
9/A  Metamodel of the IRIS-language concepts in UML-notation .... 9
10/A  An Overview of IRIS ................................... 10
12/A  A hierarchical model ................................... 12
13/A  Example for interval arithmetic in IRIS ............. 13
14/A  Time dependence in IRIS ............................. 14
15/A  IRIS-Model to exemplify the solver ................ 15
15/B  A contradiction leading to impossible bounds ........ 15
16/A  Overwriting the implicitly defined availability ......... 16
23/A  Example of a fragmented BPM ....................... 23
24/A  Categories of Bakken et al. ........................ 24

27/A  Overview of the program phases ..................... 27
28/A  The first phase .......................................... 28
29/A  Flattening a model ..................................... 29
31/A  The second phase ...................................... 31
32/A  AST for Example 32/A ................................. 32
33/A  The third phase ......................................... 33
34/A  The fourth phase ....................................... 34
43/A  Example for the arithmetic simplification ........... 43
44/A  AST for \( x = 13 + 5 - 2 * y \) ......................... 44
47/A  Example for an operator transformer ................ 47
48/A  Example result for an operator transformer ........ 48
54/A  Tailoring with local time dependence ................. 54
66/A  Proposed scheduling for the transformations ....... 66

72/A  IRIS relaxing a constraint ............................ 72
72/B  Constraint relaxation affecting behavior ............ 72
73/A  IRIS’s implicit assumption “Gewicht” ................. 73
74/A  Example of tailoring in IRIS ......................... 74
76/A  The obfuscation model preview window in IRIS ...... 76
76/B  Model preview window with an error ................ 76
77/A  Obfuscate via menu option ............................ 77
C.2 List of Tables

14/A "And", “or”, and “not” operations in ternary logic ......................... 14
38/A Interesting properties of transformations .................................. 38
73/A Transformation categories selected for the implementation ........... 73
81/A Test coverage of the proof of concept implementation ................. 81

C.3 List of Examples

3/A Behavior of a model ............................................................... 3
4/A Models which are not behavioral equivalent ............................... 4
12/A Hierarchies in IRIS ................................................................. 12
15/A Solving a model ................................................................. 15
17/A Formalizing the children relationship ..................................... 17
18/A Formalizing the interface-implementation relationship ............... 18
28/A Removing implicit assumptions .............................................. 28
29/A Constraint generation for a hierarchical model ......................... 29
30/A Simple tracing model .......................................................... 30
31/A Abstract input for the second phase ...................................... 31
32/A Identifier references by the second phase ............................... 32
44/A Applying arithmetic simplification ....................................... 44
45/A Inlining breaks impact ......................................................... 45
47/A A strength transformer ......................................................... 47
57/A Naively identify the impact of constraints ............................... 57

C.4 List of Digital Attachments

Submitted alongside this thesis are the following digital attachments (reduced for submission to OPARU):

1. Additional tools developed alongside the writing of this thesis:
   i. sirius, the tool written to automate taking screenshots of IRIS.
ii. benchmark-parse, a tool written to process the information collected by the benchmarks.

2. Raw files produced by the benchmark (csv).
C.4 List of Digital Attachments