Towards a solution for improving the performance of model transformations in Model-Driven Development

Master Business Informatics Thesis — July 12, 2017

Bart Smolders
3865789
b.t.smolders@students.uu.nl
Utrecht University

Supervisors
dr. S.L.R Jansen (UU)
dr.ir. J.M.E.M van der Werf (UU)
M. Overeem, MSc. (AFAS)

Acknowledgments

I want to thank my supervisors from Utrecht University Slinger Jansen and Jan Martijn van der Werf, for giving guidance and feedback on my research. Next, I would like to thank Michiel Overeem for the day-to-day supervision at AFAS, many brainstorm sessions and quick feedback whenever I needed it. Furthermore, I would like to thank Rolf de Jong and Machiel de Graaf who made it possible for me to do my research at AFAS. During the bi-weekly AMUSE meetings Dennis Schunselaar and Henk van der Schuur critically looked at mu research and thesis and I want to thank them for their help. Finally, I want to thank Marten, Sander, and Pepijn for the support on my research as well as the many games of table football.

Abstract

Utilizing models in software engineering is gaining popularity and there is an increasing need to generate faster. There are many approaches available to fulfill this need by leveraging the Atlas Transformation Language (ATL) model transformation language and its underlying platform to improve the performance of a generator. However, these approaches are specific for declarative transformation languages, like ATL. In this thesis we do a knowledge analysis of several such approaches because of the good availability. These are ATL on MapReduce, multi-threaded ATL, live transformations and lazy transformations. All of these approaches rely on model element traceability which is why traceability approaches are also analyzed. Our case study organization developed a generator where the model transformation are programmed in a general-purpose language. This limits the applicability of the available approaches in literature which is why a more generic approach is researched that improves the performance of model transformations. A reference architecture is proposed with corresponding documentation. This documentation describes important design decisions, required protocols and practical issues that must be taken into account or addressed when developing a generator aimed at faster model transformation. These design decisions and processes result from the performed case study where we implement approaches from the knowledge analysis. First, traceability is implement followed by partial model transformation that use the insights gathered from traceability. Both are prerequisites from parallel and incremental model transformation which are common approaches to reduce the time needed to finish a task. Finally, an experiment is performed with parallel model transformation which shows a performance improvement of 2.5 to 3.5 times.

Keywords: *Model-Driven Development, improve transformation performance, traceability, partial generation, parallel generation, incremental generation*

Contents

1 Introduction

Researchers are constantly increasing the abstraction level at which software is engineered. Sendall and Kozaczynski (2003) state that this is one of the best ways to reduce complexity in software. Model-Driven Development (MDD) is a software development method that uses abstractions in the form of models as main development artifact (Hailpern and Tarr, 2006). Large parts of the software, if not all, is generated from models. They define MDD as: "*A software engineering approach consisting of the application of models and model technologies to raise the level of abstraction at which developers create and evolve software, with the goal of both simplifying (making easier) and formalizing (standardizing, so that automation is possible) the various activities and tasks that comprise the software life cycle*". Supporting tools, such as modeling environments and IDEs exist that assist the modeler and developer, respectively. Additional advantages of MDD are increased code quality, maintainability, and productivity which are achieved as a result of automation (Staron et al., 1994; Trask et al., 2006; Weigert and Weil, 2006). Moreover, Software Producing Organizations (SPOs) may offer end-user variability to allow users to model the environment for their needs (Kabbedijk et al., 2012; Brown, 2004). Changes are applied to the application by adjusting the corresponding model and then by regenerating and redeploying. This flow presents a problem in terms of waiting time when not handled efficiency.

1.1 Problem Statement

Utilizing models in software engineering is gaining popularity and there is an increasing need to generate faster, since end-users want fast responding applications (Hussmann et al., 2011). However, organization have difficulties achieving this. Lussenburg et al. (2010) mention that industrial validation of available literature on MDD is scarce and can thus only help organization marginally. Moreover, there is currently a lack of any scientific documentation that presents information to create an optimized generator in a structured way. Such an artifact could greatly help MDD adopting organizations in building high performing generators. There is literature available on declaratively model transformations such as ATL (Benelallam et al., 2015b; Tisi et al., 2013). However, there is only very little literature available on generators built in a general-purpose language in combination with reducing generation time. The case study organization encountered this issue in their quest in reducing the generation time.

The current situation of the case organization is illustrated in Figure 1 and described in detail in Section 2.2. This figure shows the execution flow of creating an application given an input model. Executing this flow typically results in unnecessary waiting time for end-users since all transformation are executed sequentially. The total generation time may take minutes up to hours (Varró, 2015) which negatively influences the user experience (Bergmann, G., Horváth, Á., Ráth, I., Varró, D., Balogh, A., Balogh, Z., & Ökrös, 2010). For SPOs it is therefore crucial to improve this process to reduce the time required to create the initial application. This initial generation scenario is the first identified problem.

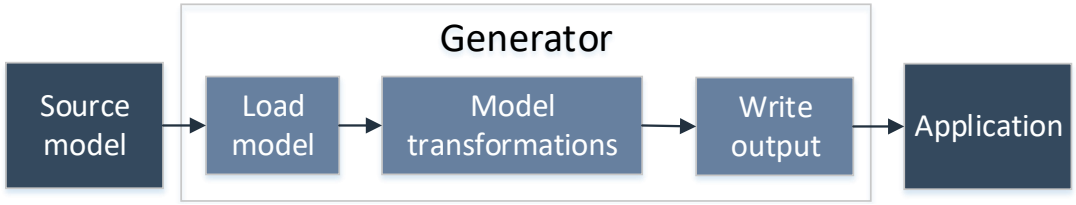

Figure 1: Generator architecture of the case study organization. The arrows show the data flow between two stages. The transformation pipeline itself is not parallel but sequential.

Furthermore, SPOs may offer end-user variability where the end-user manages the model and can thus make modifications at any time. Therefore, the end-user determines the moment when a regeneration of the model is trigged. The end-user would normally choose to directly trigger a regeneration after a model modification to apply the changes. This situation is opposed to when the organization that build the generator is in control of the model and can choose its own time slot to upgrade the applications.

Subsequent model generations form a second common use case which is called incremental generation. Selic (2003) argues that incremental refinement is very useful since it is a process that is executed far more frequent than a full generation. An implicit issue with large organizations and MDD is the high capacity of hardware resources needed for generating the applications. In case only a small portion of the model is modified there are still many resources needed when the full model is generated. This also means that a larger amount of hardware resources must be available to handle peak moments to ensure high responsiveness. Cloud-scaling cannot provide a solution because of the overhead of spawning additional servers. This issue can also be solved by using incremental model transformation.

Problem statement summarized:

> **Current MDD methods that rely on imperative transformation algorithms do not perform well on large input models. Consequently, the perceived usability and adoption of MDD on large size applications is decreased.**

1.2 Research Questions

The subject for our research is the imperatively programmed generator provided by the case study organization. The goal is to improve the performance of model transformations. To solve the problems as defined in the problem statement the following research question is defined:

RQ *How can the performance of imperative model transformations be improved?*

The following sub research questions (SRQ) are designed to answer the main research question.

SRQ 1 *What is the current state of methods that reduce model transformation time in Model-Driven Development?*

SRQ2 *How is insight gained in the model transformations in a Model-Driven Development environment?*

SRQ 3 *How can these insights be utilized to improve the model transformation performance in a Model-Driven Development environment?*

SRQ 4 *Does the utilization of the insights lead to an improved performance?*

1.3 Scientific contribution

This research provides several scientific contributions. We analyze and validate existing literature on improving the model transformation performance and how generic the researched techniques are. Literature is validated by researching the feasibility of implementing the techniques. Finally, this thesis presents an artifact as a structured document that contains a reference architecture that is designed to improve the model transformation performance. The reference architecture is based on observations and above all suggestions gathered during the research. Additionally, the artifact Moreover, this artifact is a first attempt for creating such a structured document which is in future work to be extended with observation and suggestions from other case studies.

1.4 Thesis Overview

The remainder of this thesis is structured as follows. First, Chapter 2 describe the research approach used and the research context. Then in Chapter 3 describes related work and current state analysis.. Chapter 4 and 5 research traceability and partial generation, respectively, as prerequisites for both incremental and parallel generation techniques. Afterwards, partial generation and traceability techniques are used to research incremental and parallel generation in Chapter 6. The following section combines all findings in Chapter 7 where a reference architecture is presented and, finally, Chapter 9 provides the conclusions of this research.

2 Research Approach

This section describes the used research methodology and three research methods that together form our research approach and the context in which the research is performed. The research methods are a knowledge analysis, a single case study and an experiment.

2.1 Design Science

Research in the Information Systems discipline is characterized by the behavioral science and design science paradigms. Hevner et al. (2004) define the two paradigms as follows:

Behavioral science addresses research through the development and justification of theories that explain or predict phenomena related to the identified business need.

Design science addresses research through the building and evaluation of artifacts designed to meet the identified business need.

Design science is more suitable for this thesis since the goal is to research and implement model transformation methods to address a business need. Namely, to reduce the generation time of a case study code generator used in a Model-Driven Development setting. Figure 2 shows this thesis in the Information Systems research framework of Hevner et al. (2004). The framework is composed of three parts that together result in an addition to the scientific knowledge base. The parts are environment, knowledge base and IS research. In the environment there exists a business need of one or several organizations. The knowledge base is filled with existing literature on foundations and methodologies within the context of the problem and IS research. These foundations and methodologies together form the applicable knowledge. Finally, the IS research project can be conducted based on business needs and applicable knowledge. The IS research phase is a develop/build and justify/evaluate cycle. The outcome should be applicable to the environment and is meant to fill a gap in the current scientific knowledge.

2.2 Case Study Organization

This chapter describes the context in which this research takes place. The case study company, AFAS Software, is introduced first including their product, Profit Next. Finally, the AMUSE project is described of which this research is part of.

2.2.1 AFAS Software

AFAS is a Dutch vendor of ERP software. Their headquarters is located in Leusden in the Netherlands. Besides their office in Leusden, they have offices in Belgium, Curaçao and Aruba. The privately held company currently employs over 350 people and annually generates € 100 million of revenue. AFAS offers a fully integrated Enterprise Resource Planning (ERP) suite which is used daily by more

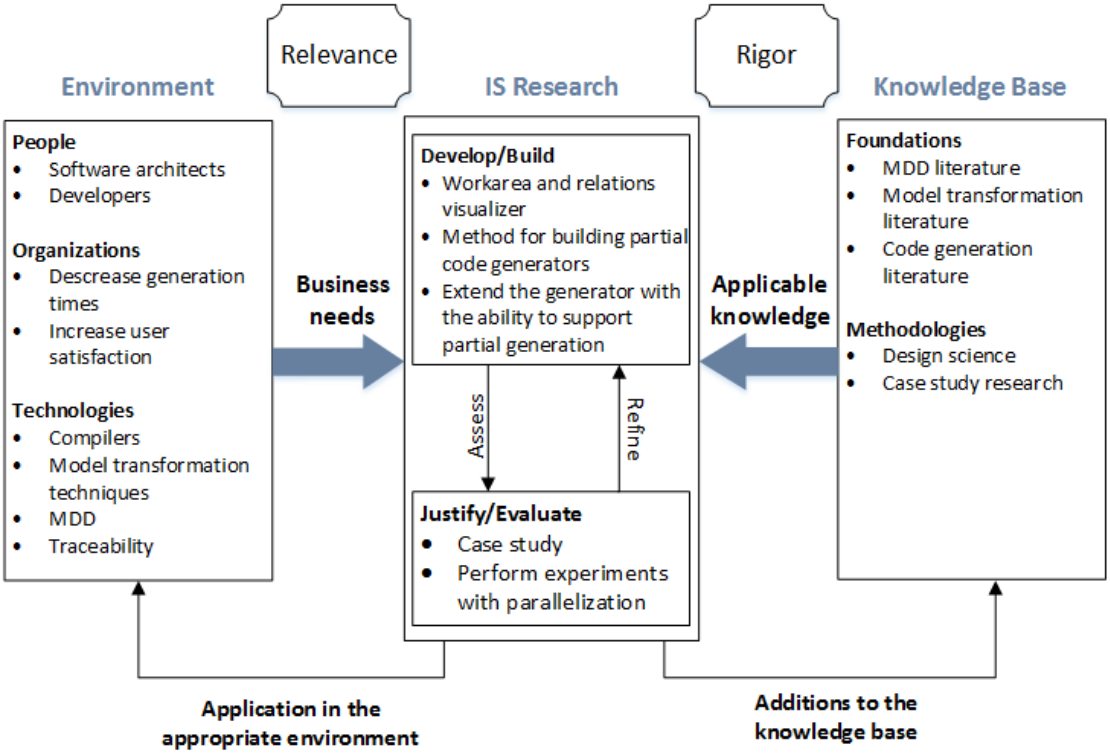

Figure 2: The research of this thesis depicted on the Information System Research Framework based on Hevner et al. (2004).

than 1.000.000 professional users of more than 10.000 customers. The ERP system is offered as a Software-as-a-Service (SaaS), called AFAS Online, or it can be hosted by the client on premise. Since 2011 AFAS Software also offers an online expenditure book aimed at consumers, called AFAS Personal.

2.2.2 NEXT

The next version of AFAS' ERP software is completely generated, cloud-based and tailored for a particular enterprise, based on an ontological model of that enterprise. The ontological enterprise model will be expressive enough to fully describe the real-world enterprise of virtually any customer. Furthermore, it will form the main foundation for generating an entire software suite on a cloud infrastructure platform of choice: AFAS NEXT is entirely platform- and database-independent. AFAS NEXT will enable rapid model-driven application development and will drastically increase customization flexibility for AFAS' partners and customers. All this is based on a software generation platform that is future proof for any upcoming technologies. Since Profit Next can already provide a fully working, not yet complete, code generator, AFAS is a great case study company for this research.

2.2.3 Case Study Modeling Language

The case study organization is designing a declarative modeling language as part of their model-driven software generation approach. This language is used for modeling an enterprises consisting of people, products and processes. The language is expressed in an Ontological Enterprise Model (OEM) and is explained by Schunselaar et al. (2016). This is the first artifact in the MDD transformation flow of the case organization. End users can use this language to tailor the application to their business needs. The OEM language support several modeling constructs that represent real-world phenomena. The five main supported constructs that relevant for our research are Entity, Role, Event, Agreement and Work area. These are explained below and depicted in Figure 3.

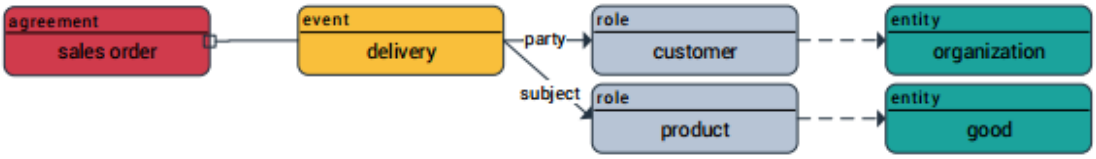

Figure 3: A simple OEM composed of the four NEXT language stereotypes

Entity An entity represents a distinguishable thing in the real world, such as an individual, a location, or a good.

Role Roles are views on entities that describe how entities can manifest themselves. When used in events, roles represent the capacity in which entities are involved in these events. As an example, an organization play the Role of a customer.

Event Events are abstractions of transactions occurring within an enterprise. Although at design-time an event is a static, timeless component, during runtime an Event can be in a future or past phase.

Agreement An agreement represents a commitment between the modeled enterprise and an external party. Agreements may apply to a specific moment or period in time.

A fifth modeling construct plays a large part in our research and that is the Work area construct. There did not exist a definition for Work area and therefore we created, together with the model designers, a definition that conforms to the boundaries of our research.

Work area A collection of Events within an organization that have a direct relation to one another, that can exist independently and have no or only indirect relations to Events in other Work area.

2.2.4 Case Study Generator Architecture

In this section the case study MDD generator which is used during our research is described. On a high level the generator architecture can be divided into three parts. The first phase load and parse all source models. Then a model transformation phase is executed that is composed of many model transformations (MT).

At the end the output is written to disk. The source models themselves are a collection of XML files and are modeled in the OEM language explained above. In the parse phase an in-memory model representation is created that contains model elements from the parsed source models. Moreover, the data structure is enriched with additional model elements and some basic behavior that are deduced from original model elements. Furthermore, the in-memory model can be traversed in the model transformation phase where more behavior is added to model elements and gradually and sequentially transformed into more concrete artifacts. The model transformations can be divided into three sections; query, command and ui. These sections correspond to the CQRS pattern used for the output application. The output created in the last phase are either DLL, JSON or JavaScript files. This set of output files together form the entire application and no modifications from the developer are required. Lastly, the entire generator is written in a general purpose language, namely C#. The case generator architecture is shown in Figure 4.

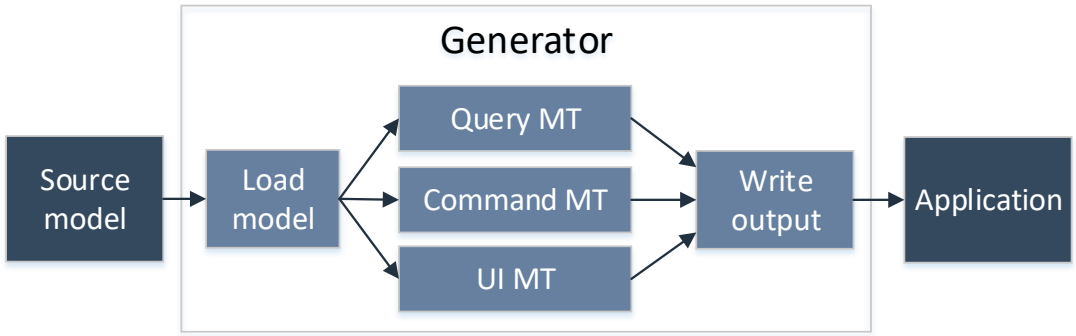

Figure 4: Generator architecture of the case study organization. The arrows show the data flow between two stages. The transformation pipeline itself is not parallel but sequential.

Rational of the Architecture

The architecture and modeling language described above are both customized for the domain the case study organizations operates in. This section explains why this architecture was chosen over standardized declarative or hybrid languages and tools like ATL. The case organization desires a solutions that works for the entire model transformation flow wheras ATL focus primarily on model-to-model transformation and not model-to-text transformation. Moreover, a significant amount of imperative programming is needed due to the complex nature of an ERP application. Furthermore, strong training of employees and thorough technical knowledge is required to turn the models into working applications with declarative or hybrid approaches as stated by van Kooten (2016). Another drawback of using third-party software in general is that developers have less or no control over applying bug-fixes to the underlying platform. This becomes especially an issue when the platform is extensively used in an industry-setting. Additionally, developers

have less control over the performance and memory management in general since this is managed by a compiler and virtual machine.

2.2.5 AMUSE

This research is part of the AMUSE (Adaptable Model-based and User-specific Software Ecosystems) project. The AMUSE research project is an academic collaboration between Universiteit Utrecht, Vrije Universiteit Amsterdam, and AFAS Software to address software composition, configuration, deployment and monitoring challenges on heterogeneous cloud ecosystems through ontological enterprise modeling.
At this moment, four PhD.'s and several master students are working as part of the AMUSE project. More information about this project can be found at https://www.amuse-project.org.

2.3 Research Design

We combined the Information Systems Research Framework by Hevner et al. (2004) with the problem-solving cycle of Polya (2014) and this results in a research design with five phases. These phases are problem statement description, knowledge analysis, solution design, case study implementation and solution evaluation and are explained next. This design is depicted in Figure 5.

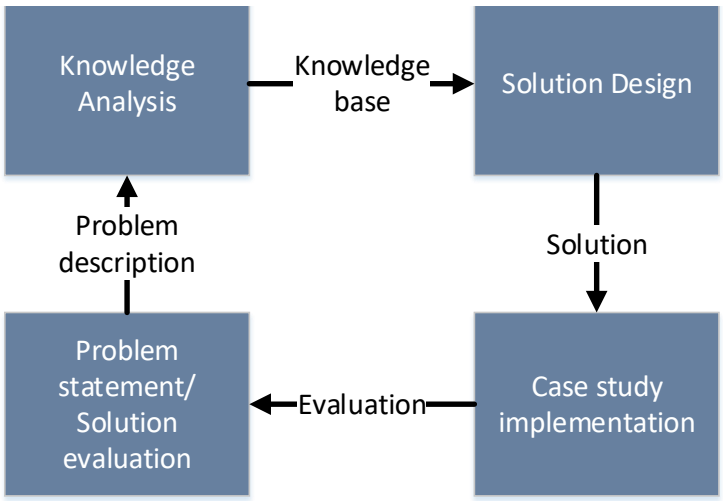

Figure 5: Information system research framework of Hevner et al. (2004) combined with the research cycle of Polya (2014).

2.3.1 Problem Statement

In this phase, a deep understanding of the problem is created. This is achieved by doing an exploratory literature search, partially based on the business needs of the case organization concerning the defined problem. This first search is conducted to find out whether there is enough reference material to find one or more candidate solutions. The exploratory literature study is done using snowballing. Scientific

literature is found by means of scientific search engines such as Google Scholar and DBLP, a database aimed at publications in the computer science domain. The deliverable of this phase is the Problem statement (Section 1.1) of this document.

2.3.2 Knowledge Analysis

The goal of the Knowledge analysis phase is to find a large collection of scientific as well as non-scientific knowledge. Non-scientific knowledge such as blog posts about incremental compilation and traceability in compilers. Moreover, existing model transformation methods are discovered, described and analyzed. This literature study is performed by using the snowball method. Several such snowball literature studies were performed for the different topics. Those topics are compilers, model partitioning and distributed model transformation, and model element traceability. Using this as a starting point new definitions and technologies are discovered. These discoveries are then used to find new scientific literature in the previously named search engines. The methods are described and analyzed in the Related Work (Section 3) and can be used in the Solution Design phase.

2.3.3 Solution Design

This phase encompasses the design of one or more candidate solution by using the information from the Knowledge analysis phase. Finding candidate solutions is simplified by looking at the case study generator. This somewhat simplifies the problem and context and make it more concrete and understandable. The generator that will be used throughout the thesis is described in Section 2.2.4. Furthermore, related literature of similar techniques and approaches of other domains where code generation and model transformation is done are used to create solutions from different aspects. At the end of this phase, one or more best fitting candidate solutions are chosen for the problem defined in the Problem statement phase.
For each of the found methods it is decided what parts of the method are included in the case study. This depends, among others, on how clear the parts are described and the properties they might rely on.

2.3.4 Case Study Implementation

The case study is composed of several smaller case studies where each research a single approach. The approaches are chosen from the related work analysis presented in Section 3. Moreover, the topic selection depends on research performed on the generator architecture which is explained in Section 2.2.4 to select a suitable a approach. For each research topic the best applicable approach is described and justified and then implemented and validated. The implementation strictly follows the presented technique by implementing all its features. The defined research topics are traceability, partial -, parallel - and incremental model transformation. The outputs of the validation step are captured in written observations about any domain specifics, opportunities and limitations. Finally, an experiment

is performed with the implemented parallel model transformation approach to validate whether an improved performance is achieved. The observations are written in the format as described in Section 2.5.

Experiment

An experiment is performed with parallel model transformation by using the partial generation implementation from Section 5. The hypothesis for this experiment is that parallelism improves the model transformation performance for initial application generation compared to sequential generation. To test this hypothesis the total generation time of the baseline is compared to the parallel version. The baseline is measured with the case generator without any modifications made for our research and the pipeline is executed sequentially. Source models of different sizes are used to measure how the generator behaves. This experiment leaves incremental model transformation (Section 6.4) out of scope since a single work area is the smallest granularity on which partial generation can currently be done.

Similar experiments were conducted by using the ATL and Map-Reduce in Benelallam et al. (2015a) and Benelallam et al. (2015b) as described in Section 3. Their approach shows a speed-up of up to 2.5 to 3 times and their results show that larger models produce higher speed-ups. The speed-up is linear in the number of computation nodes used but also shows that at a certain point it is no longer effective to add more nodes. This is expected since many small partitions result in a significant amount of overhead.

Parallel model generation is expected to add overhead (Benelallam et al., 2015b) as a result of limited hardware resources since all parallel instances run of the same machine. Adding multiple machines to divide the workload over is a solution but additional overhead is introduced in the form of network latency (Dean and Ghemawat, 2008). Overhead in the experiments can be approximated by analyzing the timings of the pipeline from start to the parallel generation point. Baseline timings can then for that phase be compared to hypothesis timings. Starting and running multiple instances of the generator at the same time forms the second part of the overhead. This would be the result of maximum CPU and/or RAM utilization. Generating model elements that are needed in multiple partial input models only have a small impact since there are only few duplicates. The trade-off between a sequential generation versus a parallel generation could become visible when the baseline and hypothesis timings are aggregated in a graph. An important note here is that the input model is created in advance by hand so no time or processing power is consumed for this process in the experiments.

Scope

The scope of the case study is limited to the generator only. Any piece of code could be used to validate any of the found methods to improve performance. An important note here is that only the concepts are validated and not the specific implementation such as using ATL to define transformation rules. Moreover, local optimizations that could improve performance are left outside of this scope. This scope is chosen to make the validation feasible in the available time. For the time

being, the case study company does not want to completely rewrite the generator and committing to a different technology. Lastly, any steps needed for deploying the generated application are also left out of scope.

2.3.5 Solution Evaluation

In this phase, the implemented approaches are evaluated as described in the Reliability paragraph. After that, the created framework as a result of the case study is checked for generalizability. This entails looking for contextual elements that are needed for a chosen technique. The goal of this comparison is to see whether industry best practices are indeed generic enough to be applied any generator. Experiments are performed to evaluate the examined and implemented techniques.

2.4 Research Plan

This research is both a descriptive and an exploratory research. It describes and analyzes the current state of multiple topics related to improving the performance of model transformations in SRQ 1. After the descriptive part we continue with the exploratory part which we research in a single case study. We start by exploring two prerequisites of improving the performance by means of parallel and incremental generation. These requirements are traceability and partial generation and answer SRQ 2 and SRQ3, respectively. We explore these topics by implementing and thereby validating a researched approach as part of the case study. After that, we can combine the implementations and results from SRQ 2 and 3 to explore parallel and incremental generation themselves in SRQ 4. During the case study observations are made about the approaches and opportunities and limitations they bring with them. Finally, all research and observations are combined into a reference architecture that answers the main research question. Observations are complemented with solutions on how to overcome the observed issue. The solution should be sufficiently generic to be applicable in other context as well.

2.5 Artifact Composition

During the two case studies observations are made which are then complemented in with suggestions in the Reference Architecture chapter (Section 7). This section shows the structure of the observations. An observation itself is composed of a unique identifier, title and description as shown next:

> **Observation X**:
> A summarized description of the observed issue.

The structure of an observation that is complemented with one ore more suggestions is described below. For better understandability we also added the topic this observation belongs to. The reason for this is that these suggestions are added in the reference architecture section and not their corresponding case study sec-

tion.

> **Solution(s) for Observation X**
> One or multiple suggestions that can be used to overcome the observed issue. Each suggestion contains a justification explaining why this is a good alternative.

2.6 Treats to Validity

Here the threats to validity are described of the related work analysis, case studies and experiment.

2.6.1 Construct Validity

Whenever possible, existing definitions of measures are used in case they are available in literature. However, when no standardized definitions can be found similar definitions are either extended to fit the current context or new definitions are provided with a proper rationale. Where possible definitions from multiple sources are compared to make sure the definitions are really applicable for the measurements and the most widely accepted definition is chosen. The main metric used throughout the thesis is the total model transformation time that the generator needs to execute the entire pipeline. The total time is calculated by the stopwatch that is implemented as part of the generator. The generator should be the only active process on the computer to ensure that two separate measures are not polluted by other active processes. Furthermore, the resulting measures are gathered from the same computer to eliminate hardware differences.

2.6.2 Internal Validity

In order to increase the internal validity, the implementations are clearly reported as well as the data collection methods used. Moreover, insight into the data analysis is given to make it reproducible for others to reviews the outcomes. The method used to compare the different implementations is documented extensively. Reproducibility is limited since the generator is completely custom built and closed-source.

2.6.3 External Validity

A literature study by snowballing is performed to gather relevant approach to improve the performance of model transformations. Therefore, it is possible that no all available approaches were found. Any libraries used to aid implementation will be open standards and open-source software to ease reproduction in a different context. The constructed framework should be abstract enough to make them applicable in other organizations. All tools that follow from this research will be made open-source and will be published on GitHub with proper documentation. The main threat to validity is that the case study is only performed at one company. A threat to generalizability our the experiment results is that the

number of elements is specific per case company. For example, our numbers are a lot lower compared to experiments conducted with ATL. The model element count itself does not say anything about the generation time or overhead without knowledge of the generator architecture. Moreover, the experiment is only conducted on a single generator and the impact of the generator architecture cannot be determined. Furthermore, models are created by hand which means that no partial model calculation is performed as part of the experiment. This calculation of course adds additional overhead to the parallel approach and is not accounted for in this experiment. Lastly, the granularity of the partitions is restricted by the partial generation implementation to a single work area per partition. A more efficient partitioning might have more and smaller partitions and should be researched further.

2.6.4 Reliability

Some reliability checks have to be done in order check the reliability of the output data. Firstly, the case study protocol is provided and checked to enable reproducibility of the results. Secondly, the implementation of different model transformation approaches are reviewed to check whether the method is indeed correctly implemented. No audit will be performed on the output of the modified generator, since it is considered out of scope to do any type and behavior checking on the resulting application. Therefore, it is unknown whether the output of the modified generator is the same as the one from the original generator.

3 Related work

This section provide related work on the topics like ATL, MDD approaches that intent to improve model transformation performance and, lastly, approaches that aim to gather traceability information in a MDD setting. Each topic is complemented with an analysis of current state of that topic and these analyses combined provide an answer to **SRQ 1**: *What is the current state of methods that reduce model transformation time in Model-Driven Development?*

3.1 ATLAS Transformation Language

ATLAS transformation language (ATL) is a rule-base language designed for model transformations in the field of Model-Driven Development, Tisi et al. (2013), and is widely used in the MDD domain. ATL conforms to specifications defined by the MetaObject Facility (MOF) Object Management Group (OMG). ATL is a hybrid that supports both declarative as well as imperative constructs. Declarative transformation definitions are preferred to raise the level of abstraction, however, imperative definitions are sometimes needed in complex cases. The declarative language is expressed in Object Constraint Language (OCL) designed to work with UML models. Furthermore, ATL transformations are unidirectional and can only work with read-only input models and produce write-only output models as presented by Jouault et al. (2006). Moreover, the ATL transformation definitions are processed by a transformation engine where input models and transformation rules are compiled to byte-code which is executed by the ATL Virtual Machine (VM). The ATL VM acts as the underlying engine that execute the model transformations and resolve variables to their concrete value.

Analysis

The current state of ATL and its tools already allows solving nontrivial problems as stated by Jouault et al. (2006). This is argued by the interest that the fast growing ATL user community shows and the increase in implementations in ATL. ATL is based on standards defined by OMG and conforms to its specifications which make it suitable for industry to built further on. ATL focuses primarily on model-to-model transformations which we consider to be a limitation. This means that it can only be applied to a limited number of transformation scenarios. For example, directly using ATL in our case study organization would not be suitable because the model-to-model and model-to-code transformation have to be addressed by separate transformation languages. Moreover, ATL itself does not support incremental model transformation. On the other hand, ATL does have strong parallelization properties as a result of its transformation formalisms and automatically create traceability at run-time.

3.2 Divide and Conquer Approach

3.2.1 Distributed Model Transformation

Distributed model transformation (DMT) is defined as the process of distributing model transformation tasks over computer clusters. Each cluster contain multiple nodes that are in charge of generating partial outputs that are later merged to obtain the full result as explained by Clasen et al. (2012). DMT is used to overcome the memory limits and long execution times when dealing with very large models of several gigabytes in size. DMT introduce topics of concurrency, task synchronization and shared data access which are considered complex. However, model transformation languages with a high level of abstraction can mitigate these issues in the form of implicit distributed execution by using declarative language constructs. Such Domain Specific Languages have strong parallelization properties that also handle the introduced complexity as stated by Benelallam et al. (2015b).

ATL-MapReduce

Benelallam et al. (2015b) propose an approach to automatically distribute the execution of ATL model transformations on top of MapReduce. Source models are partitioned into a set of partial models where the developer can control partitioning by adjusting parameters. A master node orchestrates what computing nodes are used in the model transformation process itself (map phase). Each node reads one or more partial model created before and execute an assigned map-function. Intermediate transformation results are stored locally on the node and notifies the master node that it completed its task. Then the master node search for idle nodes that can be used for the reduce phase and are assigned a reduce-function. Moreover, the location of the intermediate results that should be processes further are passed to the reduce nodes together with collected trace information in the map nodes. After that, the reduce nodes execute their reduce-function and use the trace information to resolve variables to their actual values and finally yield a result. An example of this reduce-function is a function that merges all intermediate result regarding the same source element of intermediate object instance or type.
Furthermore, the authors argue that implicit data distribution is complex when model transformations may interact with each other. Language properties for ATL are defined to reduce the amount and types of interaction between model transformations and are listed below. These properties are not necessarily specific for ATL or declarative model transformation languages but any more easily enforced for those type of languages as they are more formal. Additionally, these properties enable the decoupling of model transformation themselves and therefore independent model transformation execution.

Locality The model transformation that creates an element must also initialize its properties.

Property assignment A single-valued property created in a target model is only updated in the transformation execution. A multi-valued property can only

be modified by adding new values, but existing values cannot be deleted.

Non-recursive rule application (read-only output model) Target elements are not subject to further matches and can thus not be used as intermediate elements for further processing.

Forbidden target navigation Model transformations cannot navigate the target model to avoid assumptions on the rule execution order.

Analysis

Properties of ATL, like formalizations, are exploited to efficiently transform an input by using a cluster of computing nodes. By adapting the ATL language to fit the MapReduce distributed programming model a proven technology can be leveraged to significantly improve the model transformation performance for large and complex models. The proposed approach results in a performance improvement of up to 2.5 to 3 times on average. Not all ATL constructs are included and supported in the approach. The proposed technique does only a straightforward partitioning by equally dividing the model elements of the available map nodes. The model transformation structure is not taken into account for the partitioning even though dependencies exist to elements processed by other map nodes. Nothing is said about more complex partitioning scenarios and therefore we cannot apply this partitioning algorithm as part of our case study. Moreover, the approach is quite novel and is for now only applicable to fully declarative transformation languages.

3.2.2 Parallel Model Transformation

Tisi et al. (2013) research the scalability of model manipulation tools by using parallel execution. They mention that it is complex to implement parallel model transformation in a general-purpose language even though parallelism is a traditional way of scaling computations. The reason for this is the lack formalizations. Furthermore, they argue that ATL and the like have strong parallelization properties which is supported in the literature mentioned before. Furthermore, the default ATL compiler and virtual machine are adapted to support multi-threaded processing. The authors include task parallelism and data parallelism into their research as two common approaches to scale model transformations, both are explained next.

Task Parallelism

Each task processes the same and complete data set, but only executes a distinct set of the operations on that data. This approach works well in the case that no dependencies are introduced between tasks as a result of processing the same data set. These dependencies are introduced when tasks rely on the output created by other tasks to complete their computations as stated by Dongarra and Sorensen (1987). This concept is of course not specific for languages like ATL but also for imperative languages as is demonstrated in Subhlok et al. (1993). Transformation

rules written in ATL are highly independent from other rules which makes this approach suitable for implementing parallelism.

Data Parallelism

Data parallelism is the opposite approach in which the model is partitioned and distributed to the tasks. All the tasks execute the same set of operations on the assigned model partition. The model partitioning thereby determines how the transformation is distributed for parallel execution as explained by Zima et al. (1988). A main goal of this approach is to reduce cross-task interaction by reducing or eliminating the access of shared model elements. This is especially an interesting approach in a distributed setting where large models are processed and the communication cost between processing unit become high. This approach correspond with the ATL with MapReduce approach in the previous paragraph.

Analysis

The researched approach make use of parallelization which is a traditional way of scaling computational tasks. The authors argue that it is complex to do doing parallel model transformation in a general-purpose language and only focus on a domain specific language. This is the first approach we encountered that make use of multi-threading and has as benefit that there is no network latency introduced. Again, this approach leverage ATL properties to make parallelization more straightforward such as limited or no rule interdependency. The authors do provide a detailed description on how to deal with synchronization issues introduced by concurrent data access in parallel transformations. Moreover, the prerequisites of using ATL in combination with multi-threaded computing are explained together with functions and artifacts that are introduced. Finally, an extensive experiment was presented where a speed-up of over 2.5 was achieved for large models.

3.3 Live Transformations

Jouault and Tisi (2011) present an incremental model transformation approach for ATL. The output models are immediately updated when a change event originating from a source model is raised. This concept is called a live transformation and the presented approach builds further on existing work on live transformations. A requirement for this approach is that both source and target models are already loaded into memory.
The counterpart of this approach is called an offline incremental transformation and does not have both sets of model in memory. The source and target model are loaded right before the target model is updated. The updated source model is compared with the previous version of the model.

This incremental approach relies on two mechanisms from the ATL VM. First, while evaluating the transformations dependency information is collected. This is used to determine what transformation need to be triggered to update the target.

The second mechanism is the execution of model transformations. By default all model transformation are executed. However, with the dependency information collected a precise order and subset can be calculated in which transformation are executed.

Analysis

An efficient model transformation approach is proposed that is designed to incrementally propagate small changes in the source to the corresponding target model. The authors elaborate the modifications made to the ATL virtual machine in order to handle live transformation. Moreover, a dependency tracking approach is presented which is specific OCL expressions and the authors mentioned that their approach is not optimal. They name an alternative method for dependency tracking as future work. We consider this approach novel since advanced constructs are excluded from the research. The most important one for us are the imperative statements and is excluded because the exact impact of a change would be hard to determine.

3.4 Lazy Transformations

Tisi et al. (2011) argue that there is an increase in MDD adoption in industry contexts. With this increase it becomes clear that scalability is limited in existing MDD tools that are designed for the initial model transformation only. Moreover, these tools have performance issues when used on very large models. The authors present a lazy-evaluation model transformation approach for ATL. Furthermore, a prototype implementation is described by adjusting the standard ATL VM. In this lazy approach target elements are created at the moment when and if they are actually needed. Additionally, the authors mention that lazy evaluation is a classical method to improve performance but only under specific circumstances. One such circumstance is when only a small part of the large model is needed in one or more model transformations. The presented approach use lazy evaluation for two phases. First, the model is navigated in a lazy way and second, the target model is lazily generated. The flow of the lazy approach in combination with model transformation is depicted in Figure 6.

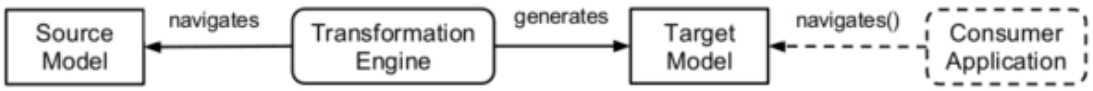

Figure 6: Lazy model transformation flow.

3.4.1 Lazy Model Generation

A target model element is created on-demand when the consumer application need that specific element. The element generation strategy is independent from the model navigation strategy. This is in contrast with standard ATL where the souce

model determine the execution of model transformations. Because of deviation of the standard ATL strategy some features are needed to support lazy generation:

- Request events to the target model have to be tracked. By tracking these events a specific transformation is triggered and executed to create the element.
- The model transformations must be able to create elements with the lowest granularity as possible like a single element or property. A low granularity maximizes the benefits of lazy transformation by only generating little outputs.
- Keep track of completed model transformations to avoid duplicate transformations. At the same time trace links between elements in the target model and their corresponding sources are created and stored. These traces can be used analyze whether derived elements are already transformed or not.

3.4.2 Lazy Model Navigation

By using a lazy model navigation strategy the access to model element is delayed and also reduce the number of accessed model elements. By using a lazy strategy the the model navigation performance is improved makes it feasible to process very large input either as a single model are by means of streaming. The model navigation strategy is used on the source as well as the target model, since the consumer application navigates the target model to start the transformation chain.

Analysis

A lazy model transformation technique is presented that enable model transformations on infinite data sets. An almost complete set of declarative ATL constructs are support by the approach. The authors mention options for future work where the performance it improved further by storing intermediate values to avoid recomputation. Moreover, a lazy OCL evaluator is needed to improve large model navigation. Furthermore, incrementality by forward change propagation is not supported at this time and the interaction between incrementality and lazy transformation is left as future work. Lastly, model transformations cannot be composed as a pipeline with this approach which is also kept as future work.

3.5 Traceability in Model-Driven Development

Software organizations use models to communicate with different stakeholders but also to manage complexity. Each of the models can be created in a different notation and these models are only connected with few relations. Multiple definitions of traceability or traceability links were found and the best suiting definition for our research is from Czarnecki and Helsen (2003): Traceability is defined as links that are created by a transformation between source and target elements used in the transformation. Relations between models are usually implicit and together with weak model integration in general introduce inconsistencies between models

(Aizenbud-Reshef et al., 2006). An example of such implicit relations are dependencies between model elements or even model transformations. Furthermore, they argue that it is necessary that traces must be generated automatically. Inostroza et al. (2014) state that some traceability research has already been performed for transformation systems like ATL, Epsilon and MOF. Huchard et al. (2006) define a trace as an ordered set of traceability links between multiple source and target models. Traceability links can be used when model transformations are done out-of-place. This means that separate source and target models exist and a concept is needed to keep both of theses models consistent. All traces combined form a traceability graph and (Czarnecki and Helsen, 2003) and Olsen and Oldevik (2007) defined traceability usage scenario in MDD that use this graph. Some of these scenarios are listed below:

Change impact analysis Analyze the impact of a change to a transformation artifact on other artifacts. Other artifacts that use the contents of a changed artifacts are possible affected depending on the type of change.

Coverage analysis Analyze whether all elements in the source model are used by at least one transformation. If an element is not used in a transformation it is either not needed in the model or a new or existing transformation has to be extend to use that model element.

Orphan analysis Output artifacts may still exist after the corresponding model element is deleted from the source model. In this case there exist no trace to the source model element and as a consequence the orphan artifact must be removed. This can b achieved by regenerating the source model to keep the model, the output and the traces consistent.

Traceability can help in these scenarios since model transformation can be very complex. MDD generator like in ATL and the case study generator are composed of multiple successive model transformations and it is therefore difficult for developers to recognize the flow of a single element and where it ends up in the output. To gather such traces a system is needed that collects trace information on the behavior of model transformations while they are executed. This concept is called a traceability framework in literature and such a framework is proposed by Huchard et al. (2006). In their research a metamodel for trace information is provided and explained how to collect trace information. They use Kermeta to write model transformations which has both object-oriented and model-oriented constructs and use imperative structures. Moreover, they note that it is difficult to create traces for these model transformations because of the imperative syntax of the Kermeta language. The proposed approach conforms to three defined features:

1. Generic traceability items.
2. Trace serialization.
3. Simple transformation from traces to a dot graph for visualization.

Furthermore, the following definitions are provided:

- **Object** *An Object is the most general element and forms the base of all other elements in the Kermeta language.*
- **Link** *A Link references one source and one target* ***Object****.*
- **Step** *A Step contains the traces of a single transformation and are composed of multiple* ***Link****.*
- **Trace** *A transformation trace is a bipartite graph. The nodes are partitioned into two categories: source nodes and target nodes.* ***Step*** *can be chained as an ordered set to from a transformation chain trace.*

Jouault (2005) also research how traceability information can be collected by adding pieces of code to a program. However, no specifics on the structure of the traceability information is provided.

In the MDD approach it possible to create trace information explicitly or implicitly in the transformation specification (Olsen and Oldevik, 2007). Implicit means that some underlying platform creates the traceability information automatically during the execution of the transformation. Explicitly adding trace code can be done in two ways. First, pieces of trace code have to be added into the transformations themselves. Second, by using a higher order transformation that analyses the model of the transformations. It is important that traces are created for all transformations since this is required to provide end-to-end traceability. The impact of a change for example can only be reliable determined when full knowledge of traces is available. Moreover, the authors provide an alternative traceability to the one presented previously but is specific to model-to-text transformations in the MOFScript tool.

Analysis

The work of Huchard et al. (2006) present a simple yet complete traceability framework. A language independent trace metamodel is presented together with used definitions and features added to enable traceability. As the authors state it is necessary to insert trace generation code and the paper provide complete code examples of both the use in a transformation and classes and methods needed for the creation and collection. Because of the imperative transformation language used and complete code snippets we can use this approach in our case study.
The traceability approach described by Olsen and Oldevik (2007) in only briefly elaborated since the authors focus on a range of traceability scenarios. The paper itself present some code, trace scenario examples and a traceability analysis prototype but all contain specifics of the used MOFScript tool which greatly limits generalizability. Since trace generation is implicit in MOFScript which means that all references and transformations can be traces, however, no real explanation is provided on how to collect trace information. Therefore it is not applicable in our case study.

4 Traceability

This section researches traceability in MDD in order to answer **SRQ 2**: *How is insight gained in the model transformations in a Model-Driven Development environment?* Traceability is the process where traces or dependencies are collected from model transformations and the model itself. All collected dependencies together form a graph which provides insights that is needed to partition source models and to determine what target elements need to be updated given a source model change. We first describe the case study method and traceability technique used for the case study. Then the case study itself is explained.

4.1 Huchards Traceability Framework Implementation

In Section 3 several traceability techniques are identified. For this case study the approach proposed by Huchard et al. (2006) is selected as is referred to as Huchards traceability framework. This approach is considered best applicable because of the language independence. Before actually implementing this framework model transformations are analyzed like the structure and references to other transformations.
After the initialization phase Huchards framework can be implemented by build the logic to create and store traces. The main logic is place inside a helper class that contains a repository that stores created traces and a method for creating the trace itself. We used a alternative setup for the traces and used different definitions:

- **Trace model.** *A Trace model is a directed graph and contains all created traces.* This corresponds to the Trace definition of Huchard et al. (2006).
- **Trace.** *A Trace is an object with references to one source TraceElement and 1 or more target TraceElement.* This definition resemble the Link definition defined above.
- **TraceElement.** *A TraceElement is a simple object in the helper class and is created by converting a C# object. This element has an identifier property and a object type property.* This corresponds to the Object definition. The create trace method in the helper class converts the source object in a model transformation to a *TraceElement*. The type property is used solely in the visualization explained later.

We omitted the Step concept because for now we have enough information to construct the *Trace model* and do not need to know in what model transformation which trace was created. Moreover, the object type globally indicates what transformation was performed or otherwise the *TraceElement* can be extended to add more transformation specific details. We deviated from the structure and definitions defined Huchards framework to make the trace model more compact. We use the source element as repository key and must therefore be unique, the corresponding values are string identifiers and are thus lightweight. Furthermore, we do not serialize the entire Object where we argue that this is not necessary

since object identifiers should be sufficient for analysis. Moreover, object serialization result in circular dependencies in our case generator.

For the trace creation itself code must be added to the model transformation code and this process is called explicit tracing. Since model transformations are intertwined throughout the code it can be difficult to recognize a single atomic model transformation. Therefore, it is difficult to choose the locations in the code where trace generating code should be placed. Furthermore, while adding trace generating code we noticed that it soon became arbitrary where this code was added. The approach we followed was to track the flow of a single source element through the transformation pipeline. At the point where the element is converted to another object type(s) we added trace generating code. In literature we did not came across a structured method on how to implement trace generating code into the model transformation code. This is especially an issue when trace code is added into an existing generator, otherwise the trace code could directly be added while programming the model transformation logic. In both implementation scenarios it would still be an issue to determine what object-to-object transformations must be complemented with traceability code. Secondly, the approach requires significant work to trace all relevant model transformation which becomes increasingly complex for larger generators with over 200.000 lines of code. This observation is summarized in **Observation 1**.

> **Observation 1**: Adding trace generating code to a MDD generator can be complex and time consuming. Transformations written in an imperative languages can be complex that also make it difficult to correctly add trace code. Declarative transformations are mostly one-to-one transformations which are straightforward to trace and can usually be handled by the underlying platform.

After having implemented trace generating code between the model transformation code it is important to keep the trace code up-to-date (**Observation 2**). Model transformation can be adapted over time and the corresponding trace generating code must be kept up-to-date. Such an adaption can be a straightforward deletion or can be an operation where some logic is changed regarding the model transformation where the trace object themselves keep their meaning. This process becomes more complex when the functional meaning of a traced object is altered. The developer would in this case have to determine whether the trace is still needed and/or whether additional traces have to be created in case a trace chain is interrupted.

> **Observation 2**: It is important to keep the trace code up-to-date to ensure that a trace model can be created and that captures all traces between model transformations. The trace model is considered incorrect when it does not represent all traces that should exist given the source model and model transformation and can in that case not reliably be used for partial generation.

For efficiency purposes we only want to gather useful trace information (**Observation 3**). We noticed that it is hard as a developer to determine which transformations have to be traced. There are situations where transformations do not provide additional information regarding traceability. This might be the case for a chain of one-to-one model transformation where no new complex dependencies are introduced and the existing dependencies can be induced.

> **Observation 3**: Eliminate redundant traces in the trace model. For the sake of fast generation times it is useful to not trace all model transformation when this is not necessary.

However, it can be difficult to recognize this type of transformation since extensive knowledge is needed about the entire flow of the generator. This becomes increasingly complex as the size of the MDD generator grows. Nonetheless, developers can be aided in this process by a visualization of the trace model which makes it easier to recognize redundant traces. Moreover, by analyzing the trace model a program can hint redundant trace links. A precondition of using these supporting tools is that a complete trace model must be created. This is needed since automatic analysis is only reliable when full knowledge of the domain is available, otherwise invalid conclusion could be made. This complete trace model can then based on analysis be slimmed down to reduce the amount of collected traces. However, this would lead to an impasse when model transformation in the generator are added, edited or deleted. These operation would likely alter the trace model and again the complete trace model is needed to determine whether the trace model is affected and what trace an safely be removed from the model.

Lastly, we store the trace model in a file to make it reusable during model evolution. Therefore, we researched what format can efficiently store the trace model since no literature could be found on which format can best be used to store the trace model in and resulted in **Observation 4**. XMI is de facto standard in declarative model transformation such as in ATL where all models are expressed in XMI. In the case study we serialized the in-memory trace model to JSON since the available XML serializer could not serialize our complex data structure. We argue that JSON is a good format to use since it is more compact than XML and might be faster to serialize and deserialize compared to XML. These properties should make it more suitable format to make sure that trace model export and import do not slow the pipeline more than it should.

> **Observation 4**: No literature is available on what format is best used to store the trace model in. Several common format, like XMI and JSON, can be used and each provide benefits as well as limitations. XMI is more expressive where JSON would normally yield a smaller file size. We consider this an important matter since reading and writing an additional model next to the source models themselves imply also additional computation time.

4.2 Case Study Evaluation

A final statement on the observations of this case study is that explicit tracing is complex and little guidance is available. More specifically, there is only little literature or industry best practices available to tackle the observed issues. Furthermore, implementing trace generating code in a generator with no or little formal structure, like model transformations programmed in a general-purpose language, is difficult. This research topic is often omitted in existing literature because of this complexity.

5 Partial Generation

This section researches partial model generation in order to answer **SRQ 3**: *How can these insights be utilized to improve the model transformation performance in a Model-Driven Development environment?*. First, we explain why partial generation is needed as a brief recap. Secondly, this section describes the research on model partitioning in Section 5.3 that utilize the gained insights from the traceability section. Lastly, a partial generation technique from literature is implemented and researched whether the technique is generalizable to other contexts in Section 5.4.

5.1 Partial Generation Rationale

Partial generation can be used when only a partition of a whole model must be generated. Partial generation is used in areas where very large models must be processed efficiently. Partial generation is just like traceability a prerequisite for both incremental and parallel generation as explained in the Introduction. Multiple partial generation instances can be executed in parallel to reduce the total time needed to complete the transformation.

5.2 Data Parallelism Justification

Parallel generation (Tisi et al., 2013) or distributed model transformation (Benelallam et al., 2015a) can be achieved in multiple ways. The first approach is achieved by passing a partial model to each generator instance, also called a node. No changes to the generator itself are made to process input models in parallel and this approach is called data parallelism. However, there is some application logic needed to partition the model(s) which is not elaborated in detail in the literature presented earlier. The second approach distributes the transformation logic over multiple nodes by statically analyzing the model and transformation rules. Each node works on the entire source model but only the subset of transformations assigned to the node process model elements. This is called task parallelism and both data and tasks parallelism are presented in Section 3.2.2.
Additionally, Distributed Model Transformation, described in Section 3.2.1, is a technique that leverage the computing capacity of one or more clusters in the cloud. Moreover, the approach is fully dependent on transformations written in ATL.

Based on the insights gathered (Section 4) on the case generator architecture and the literature analysis the **data parallelism approach** is selected for implementing in the case study. The rationale behind this decision was that no modifications to the generator itself are necessary. Moreover, task parallelism is not feasible for our case generator. The entire generation is built in an imperative language with hundreds of thousands of lines of code with no clear separation between different transformations. It was not always clear where one transformation ends and another one starts. Some transformations are merged into one and this

tight interconnection makes it unfeasible to try and parallelize model transformation to separate tasks as in task parallelism. Furthermore, it is not feasible to research the Distributed Model Transformation approach further since we would have to write the case generator to ATL.

5.3 Model Partitioning Implementation

Model partitioning, which is needed for data parallelism, is not straightforward. The reason for this is that static analysis of the transformation is very complex for imperatively written transformations. Benelallam et al. (2016) supports this claim by outlining that model partitioning is very challenging because of the many dependencies between model elements combined with complex transformation rules. Moreover, literature is provided on model partitioning for ATL transformations and is described in Section 3. However, there is insufficient applicable literature on the topic of model partitioning for non-declarative transformations. Furthermore, they both focus on the domain of declarative transformation languages, i.e. use formalizations to reason about model partitioning and transformations. These papers cannot directly be validated as part of the case study since we do not possess such formalizations in the case generator. Consequently, we can only research model partitioning of the source model of the case organization.

For this part of the case study we explore the possibilities to partition or prepare the source model of the case organization for partial or parallel generation. Only the source model is analyzed since the transformation themselves cannot directly be analyzed in a formal way. The case organization model can be partitioned by dividing the model on the Work area construct. By definition this is a suitable model element and results in a high granularity. As a reminder we defined the Work area as:

> *A collection of Events within an organization that have a direct relation to one another, that can exist independently and have no or only indirect relations to Events in other Work area.*

This results in partitions with elements belonging to a work area and a separate partition that contains all model elements that do not belong to any work area.

We visualized the model itself to get insight into the dependencies between model elements of different work areas. We abstracted all underlying model elements and only show work area elements and the dependencies between them including type of dependency. The resulting graph is visualized in Figure 7. This graph shows 21 partitions (the actual names of those partition are not important here); there are 20 red dots which represent work areas on which the partitioning was done. In the center there is a gray dot which represents the 'remainder' partition in which all model elements are placed that do not belong to a work area. As can be seen from the figure there are only few dependencies between work areas which would indeed make it a suitable element type to partition the model on. Moreover, there is a significant amount of dependencies to and from the 'remaining' partition. The reason is that this remaining partition contain many

general elements that can be used to specify behavior on Event elements. Due to the case generator architecture it was necessary to load the entire model into memory for each generator task. This is needed to make sure that all objects in the generator have access to model element they reference. Further, entities and roles have to be in the same partition since these element type are more tightly connected. These partitions are fine for the purpose of the case study and enable us to do this partial generation case study.

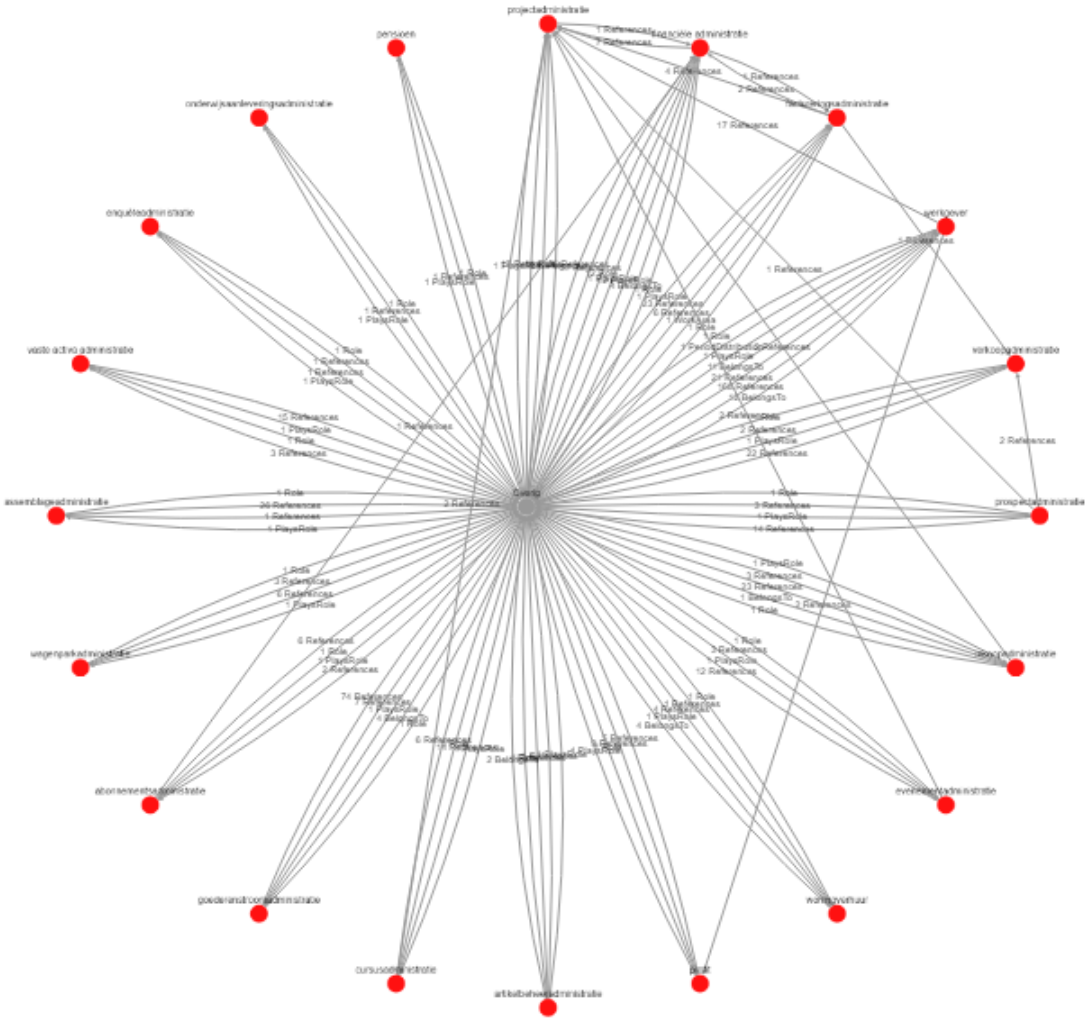

Figure 7: Work area dependencies of the case organization source model. This graph is used to gain insight into existing dependencies and what logical partitions can be made.

From the visualization we can conclude that the case input model was not designed with partial generation in mind. There are dependencies between work areas, moreover, there are also many internal dependency relations within a single work area. This makes it difficult to partition the model elements into equal partitions and this problem is recorded in **Observation 5**. This is generally desired because equal partitions result in the highest time reduction when doing parallel

generation. That is, all partial instances should have more or less the same running time without any outlier which delays the generation and deployment flow. It must then, however, also be the case that the model element type distribution is approximately equal for the partitions since the generation time per element type can differ significantly. For this research we only used a single input model, the model-driven engineering development process becomes much more complex when multiple models are added with different levels of abstraction as stated by France and Rumpe (2007). Additional complexity is introduced when versioning, refinement and dependency relationships exist between these models.

> **Observation 5**: The structure of the source model significantly affects the resulting partitioning. A source model can usually be seen as a graph which means there are dependencies. The number of dependencies is ideally kept to a minimum to partition the model into equal parts. This is turn also reduce the number of overlapping elements in partitions.

Furthermore, the following questions were encountered during this case study for which we nor literature could provide any answer to. The questions are captured as separate observations and listed below.

> **Observation 6**: No guideline or protocol could be found to determine what dependencies are relevant for a given change. Related to that; how can this be leveraged to decrease the number of elements that need to be processed?

> **Observation 7**: No suitable element type or unit is found that is best used to do model partitioning. Furthermore; what properties does such an element type need to be effective and efficient for partitioning.

5.4 Partial Generation Implementation

For the model partitioning we chose a simplistic approach implemented as additional steps inside the transformation pipeline. Furthermore, we used the gathered insights on the generator architecture to find a good location from where the pipeline can continue with partial generation and this is depicted in Figure 8. The partial generation filter (PGF) filters the source model on a specific element type as described before. After this point the architecture is analogous to the data parallelism approach in which a model partition is transformed by the complete set of model transformations. This point is ideally located as early in the pipeline as possible. The earlier such logic can eliminate model elements the better since it reduces the number of in-memory objects and output artifacts created. As a result, this elimination also reduces the generation time and unnecessary memory usage.

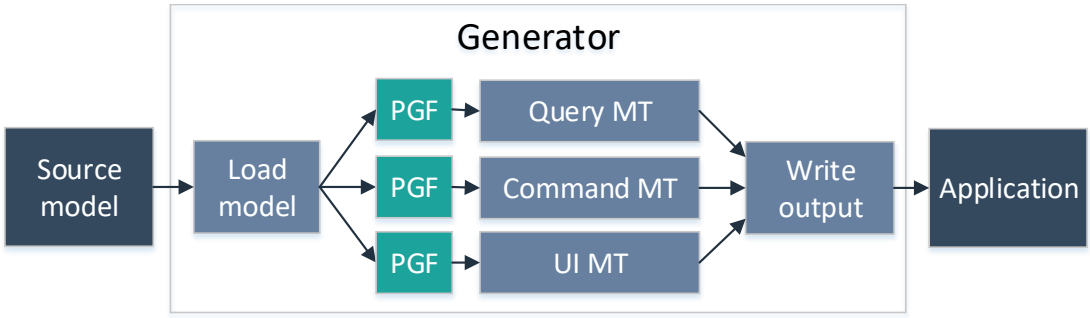

Figure 8: Generator architecture of the case study organization extended with our partial generation filter (PGF). The arrows show the data flow between two stages. The transformation pipeline itself is not parallel but sequential.

We slightly deviated from the data parallelization approach by not partitioning the source model before passing the model partitions to the nodes. A single partition could not be transformed on its own since there are references to elements outside the current partition. As a result, it can be concluded that partial transformation must succeed the model loading phase. Therefore, the PGF modules are added after that phase which is a filter to reduce the number of model elements that are to be processed by succeeding model transformations.

In our case study we encountered an issue that is a consequence of partial generation with a generator designed for sequential generation. Our sequential generator assumes the availability of all objects but is not the case for partial generation. An example of this is the phase where in-memory objects are compiled to a software component and is captured in **Observation 8**. To be able to compile a subset of the generated artifacts much more dependency analysis has to be performed. This means besides taking design-time and generate-time dependencies into account to also include run-time dependencies. However, this would make the partitioning phase even more complex. Moreover, the size of partitions will grow by significantly since much more elements are needed in the partition. This in turn reduce the number of parallel instances that can be used and increase the time needed to analyze the trace model for partitioning.

> **Observation 8**: When the MDD generator generates source code for a compiled language it cannot be compiled directly by that task itself. There are usually compile-time dependencies to artifacts that are not created by the current tasks and thus do no exist yet.

Moreover, a fundamental issue was encountered on how partial generation can be implemented and how to do it efficiently. The model partitioning algorithm require information that is needed to partition on. Other settings might be a set of model element to process or a toggle that indicates whether the entire model has to be generated. This information result from either pre-configured settings or from intermediate generator artifacts. This information could either be globally available or passed through the transformation pipeline up to the point where it

is needed but no best practice is found and resulted in **Observation 9**.

> **Observation 9**: Make partitioning properties or information available to the partitioning algorithm. No efficient strategy is found for any architecture.

We implemented the partitioning algorithm as the first phase after the source model loading model. The reason for this is that all model elements should be available during generation since there might be dependencies to elements outside the created partition. This in turn is a consequence of our implemented filter which can only partition in a simplistic way that does not explicitly take into account any cross Work area dependencies. Because of this implementation in the available architecture there are multiple locations which use the partitioning code and results in **Observation 10**. The downside of this approach that the same code has to be maintained and kept consistent multiple times. Currently, our approach is very simple and partitioning very light-weight and thus does not have a large impact on generation time or resource utilization. However, this approach becomes an issue when an entire traceability model has to be analyzed multiple times.

> **Observation 10**: An early phase or a sufficiently high abstraction must be chosen to do the partitioning. Otherwise, partitioning code is spread throughout the model transformation pipeline.

5.5 Case Study Evaluation

The observations indicate the novelty of partial model transformation in the domain of imperative model transformation. As is the case in our traceability domain there is only little literature or industry best practices available to tackle the observed issues. Guidelines or protocols are needed to handle the problems which is lacking.

6 Parallel and Incremental Generation

In this section the previously researched traceability and partial generation are combined to research both parallelism and incremental generation. By doing this we can provide an answer to **SRQ 4**: *Does the utilization of the insights lead to an improved performance?*. We start with parallel generation where we perform an experiment in Section 6.3. After that, we research the implementation of incremental generation in the case study generator.

6.1 Parallel Generation Rationale

Issues might arise regarding limited memory and high execution times when dealing with large and complex models as mentioned by Benelallam et al. (2015b). Tisi et al. (2013) state that parallelism is a traditional way of scaling computations and is already extended to the context of MDD. Moreover, Benelallam et al. (2015a) show that the generation time can be greatly reduced by using multiple tasks over which the work is divided.

6.2 Data Parallelism Continued

Research on parallel generation continues with the n our partial generation research and continue with the data parallelism approach. As was already stated in the related work section no modification to the generator itself is needed to use the approach for parallel model transformation. However, to be able to use our case generator in a parallel setup we had to duplicate the source model for each parallel task to avoid file locking conflicts. We define a parallel task as a process that use the generator executable with a set of arguments that runs in parallel with other instances of the generator. These argument are, among others, the location to the source model, the partition to generate and the output location where created artifacts are written to.

All partial generation tasks that are executed in parallel are generating their own output artifacts, which are files in our case. However, in the case of duplicate filenames, for example, the actual contents of the files has to be merged. When this is the case each partial task needs a dedicated output folder to avoid file locking conflicts while writing to disk. Merging here comprise two operations, first the separate output folders must be copied to a single folder which is used by the application. Secondly, in case partial files are produced a merge operation must be executed. This merge often requires domain information to know how 2 or more files are to be merged into one. A merge as in version control systems does not work in this case since two partial files are not actual versions of each other.

> **Observation 11**: Partial output artifacts created by separate partial tasks must be merged to form a complete application. The merge operation of these files require domain knowledge such as what the content of the artifacts represent and how it should be merged.

6.3 Parallel Generation Experiment

The goal of the experiment is to prove a reduction of the total generation time and possible overhead by using parallelism. Additionally, this chapter describes in detail how the experiment is conducted and how the results are measured and analyzed.

6.3.1 Initialization

Experiment hardware

All measurements are performed on a virtual machine (VM). The VM is installed on a native hypervisor together with two other VMs. The hypervisor runs on a host server with the following specifications:

- Microsoft Windows Server 2012 R2 Standard
- Processor: Intel Xeon CPU E5-2620 @ 2.40GHz v3, 8 Cores, 8 Logical Processors. L1 cache: 8 x 32KB. L2 cache: 8 x 256KB. L3 cache 8 x 15MB.
- Memory: HP DDR4-2133MHz 16GB.
- SSD: HP LK0800GEYMU 800GB.

The VM we used has the following specifications: 8 CPU cores, 16 GB RAM and 70GB SSD. Where possible all programs not needed for running the operating system or conducting the test are shut down. This is done to eliminate unnecessary load on the system. After every test there is a cool-down period to make sure any CPU activity of the previous test does not affect the next test.

Input Models

For the experiments we created custom input models. We create a single model for every test case and is composed of a predefined number of model elements as specified in Table 1. This table specifies the model of one test, other tests have the same ratios except for the number of Work areas. The tables for the other models are listed in Appendix A. As can be seen from the table, this model consists of nine work areas. All events are evenly distributed over the work areas. These separate work areas form the lowest granularity on which parallel generation can be done in the hypothesis test.

The model element distribution of these models are based on the existing model of the case study organization. We used the ratio of entities, roles and events (1:6:6) for designing the first model version. These element types form the foundation of the model and add most behavior to the output application. Any other element types are left out for this experiment. We then increased the amount of each element type to create a model that was comparable in size with the model the case organization currently uses. Since only Events are partitioned in Work areas there would be a large separate partition in which the entities and roles are

places. This partition would then for a large part influence the total generation time averages. For this reason we choose to adjust the ratio to make sure that all partitions that are to be generated in parallel are of equal size. This results in a more or less ideal scenario where the highest speed-up can be demonstrated and the resulting ratio was set to 1:1:18.

Table 1: Component distribution per test case.

Test case	# Entities	# Roles	# Events	# Work areas	Total
1	5	5	90	9	109
2	10	10	180	9	209
3	15	15	270	9	309
4	20	20	360	9	409
5	25	25	450	9	509
6	30	30	540	9	609
7	35	35	630	9	709
8	40	40	720	9	809
9	45	45	810	9	909
10	50	50	900	9	1009

The exact model structure of all models that are created for the experiment can be found in Appendix A and have the same distribution as described above. The models are designed to be as realistically as possible while still able to be generated in parallel. A realistic ratio is desired since the type of model element influences the time needed to generate it. The number of attributes per model element is kept constant as well as the number of model elements per work area. This is done to keep the number of model elements per model the only varying factor in the test cases of the experiment.

Parallel Generation Comments

Parallel generation is accomplished by using the partial generation approach discussed in Chapter 5. In the current implementation each partial instance generates to its own output folder due to duplicate file creation and with that file locking conflicts. These outputs must be merged to form a working case study application. This situation is the result of the implementation and programming language used for the generator.

6.3.2 Experiment Structure

The experiment is executed by conducting ten sub-experiments and each experiment consist of a baseline and hypothesis test which are described below. Each test is composed of ten test cases where every subsequent test case uses a model that contains a larger number of model elements. Each test case is repeated ten times (called a run) and the averages will be used for analysis. All steps of the tests themselves are automated in PowerShell. There is a cool-down period after a run of three minutes. In this interval period the hardware utilization will be restored to normal levels to make sure subsequent runs do not affect each other. The generator itself creates a timings file that is automatically generated while

generating. This file contains the timings of each step in the model transformation pipeline of the generator and is used for later analysis.

Baseline Test

A baseline test is executed for each test case by generating the complete model with the standard generator. This version of the generator does not contains any changes made for this research like partial generation or traceability information collection. This generator version is suitable for baseline measurements since it is made before our research started. Moreover, its properties were used to formulate the business problem of the case study company.

Hypothesis Test

The hypothesis test uses n+1 input models per test case, where n is the number of Work areas. These tests use the partial generation implementation from Section 5 to run in parallel.

6.3.3 Analysis

Each sub-experiment results in two sets of ten total generation timings, one for the baseline and one for the hypothesis. The total generation time for the hypothesis test is the task with the longest running time. For the analysis itself we use the averages of the total generation timings. The graph with ten parallel instances is shown in Figure 9. Figure 10 compares all the parallel results from tests conducted with the large models. The detailed data and corresponding graphs can be found in Appendix B. These charts depicts a time reduction given a certain number of model elements by using parallelism. On average we achieved a generation time reduction of 2.5 to 3.5 times. This is slightly better than the results achieved with ATL on MapReduce as explained earlier. However, it must be said that our models are not real world models and in practice such an equal distribution would be hard to accomplish due to model and transformation dependencies.

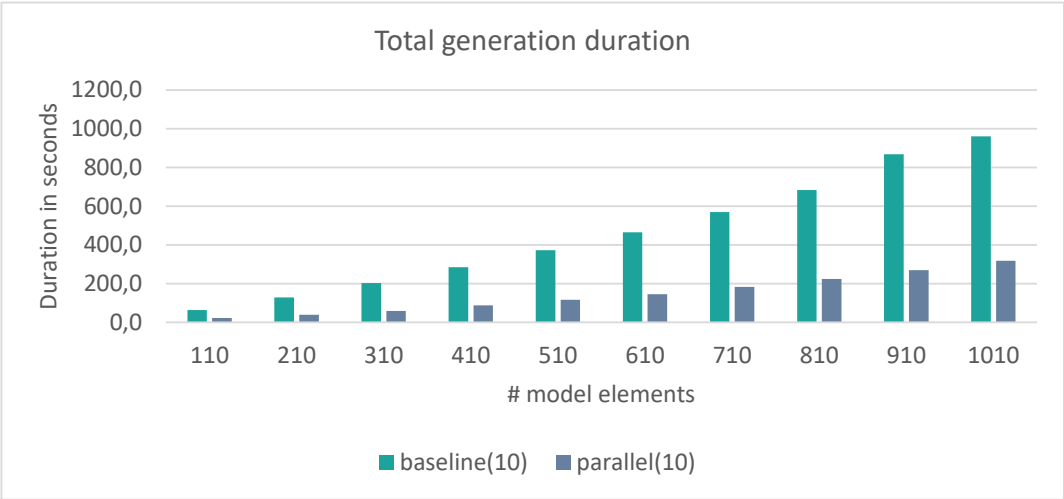

Figure 9: Generation duration chart with standard model generation compared to generation with 10 parallel generation instances.

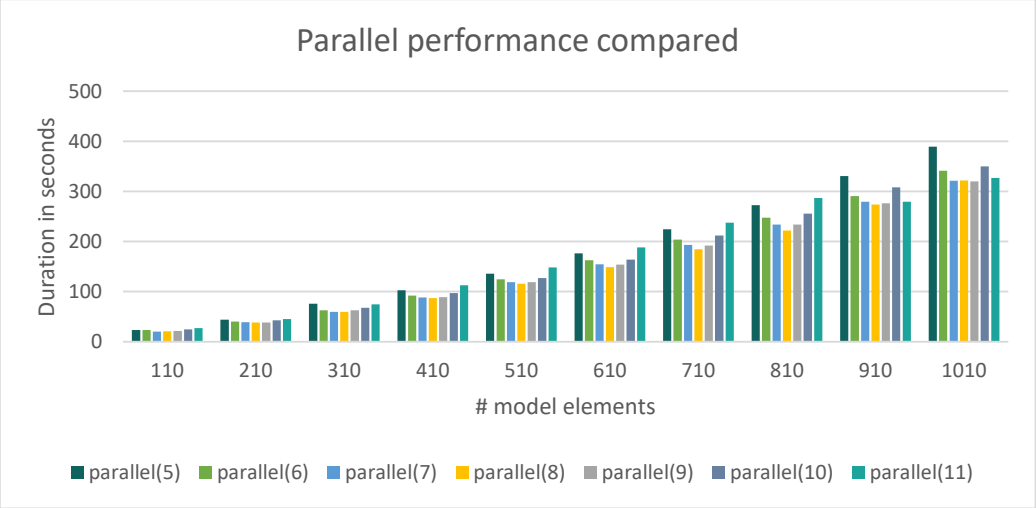

Figure 10: Parallel generation durations of 5 to 11 parallel tasks combined.

Moreover, overhead is introduced by, among others, running multiple generator tasks at the same time, as described above. However, the overhead is not directly visible from the results since we manually created the models and no partitions are computed. Nonetheless, the experiment results do show that the parsing operation becomes slower when generating in parallel compared to the baseline and is summarized in **Observation 12**. The disk is normally a common bottleneck but has only 10% utilization during the parsing phase. The probable bottleneck in this case would be the CPU which is nearing 100% utilization when multiple generator tasks are running at the same time. This observed trend holds for all transformations before the introduction of our partial generation construct. These transformations take up to 2 to 3 times longer in the hypothesis tests compared to

their sequential counterpart. The experiment result that supports our observation is depicted in Figure 11.

> **Observation 12**: Loading multiple source models at the same time adds transformation time to the model loading phase.

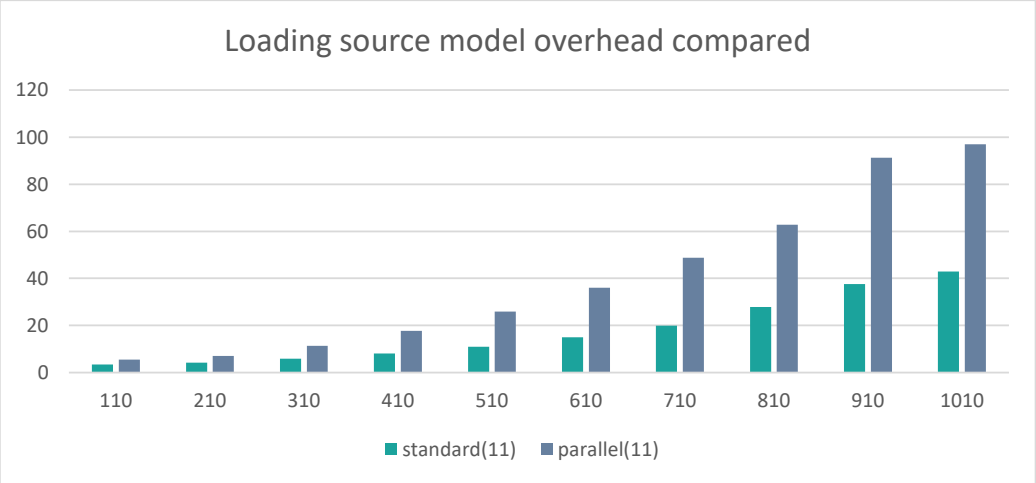

Figure 11: Source model loading overhead with 11 parallel task compared to sequential model transformation.

Lastly, the separate steps in the pipeline can be compared where timings from the parallel tasks are aggregated. The standard generation is again used as the baseline and durations from the partial generations are compared against that. The aggregated timings compared to the baseline timings shows the overhead of parallelization as a whole.

6.4 Incremental Generation

6.4.1 Incremental Generation Rationale

When a user can make structural changes in a given environment the model for that application changes. In the case of MDD, the application can simply be regenerated in order to comply with the wishes of the user. However, it is not efficient to regenerate all code when only a small portion on the model changes Varró (2015). Regenerating the entire model results in unnecessary waiting time for the end user. Efficient generation means only regenerating that part of the model that changed combined with possible dependencies (Hearnden et al., 2006). The part that changed between two model versions is called a delta. By only applying transformation rules that match one of the model elements in the delta, a performance gain can be achieved. This process is called incremental transformation and is formally defined by Czarnecki and Helsen (2006) as; the ability to update existing target models based on changes in the source models.

6.4.2 Live Transformation

No suitable approach was found that could be implemented as part of a case study. The live transformation approach from the related work section is expected to potentially achieve the highest performance improvement since both the source and target model are continuously kept in-memory. This means that no read and write operations haves to be performed. In our case study this would reduce the total generation time by 36% for large models. However, we could not implement this approach due to the lack of a traceability model and the infrastructure needed to keep the application in memory is not readily available. The traceability case study provide us only with a small set of traces for only a few model transformations. It was too time consuming to add trace generating code throughout the complete generator as is elaborated in **Observation 1**. Despite not being able to implement an incremental model transformation approach we could reason about problems regarding the implementation and use an incremental transformation approach. The first issue we foresee in doing incremental generation is the process of calculating the delta of two model versions. In **Observation 12** we observed that is not desirable to load and parse multiple models at the same time. At some point a delta of two models must be calculated to use in incremental transformation. These two models are the updated model and the previous model version on which the changes are applied. We consider that is it useful to optimize this process since it would take unnecessary time and hardware resources to load and parse two complete sets of source model each time (see **Observation 13**).

> **Observation 13**: Loading new and previous model to calculate a delta is time consuming. This delta-calculation approach is best avoided when possible to prevent introducing transformation overhead.

A second problem we expect regards the updating process of the trace model after the model is adapted which in turn invalidates the existing trace model (**Observation 14**). Model dependencies can be analyzed from the model itself but additional dependencies are introduced in the generator itself. The trace model can be updated by transforming the new model which automatically creates a new trace model as a result. However, this would defeat the purpose of using a trace model and incremental transformation since the transformation already created the complete application. Moreover, there is no applicable literature available that directly addresses this issue in our domain.

> **Observation 14**: Efficiently creating a traceability model from an updated model is not straightforward. A native approach would execute the entire model transformation pipeline with the new model to collect all relevant traces for that model. However, this approach does not use any benefits of the incremental model transformation approach.

7 Reference Architecture

This chapter describes our reference architecture that can be used to design a MDD generator tailored to fast model transformation. The references architecture is based on observations made in the case studies described in the previous chapters. The reference architecture answers the main research question: *How can the performance of imperative model transformations be improved?* and is presented in Figure 12

Furthermore, in this chapter the observations made previously are complemented with suggestions on how to deal with the observed problem. Not all observations could be complemented with one or more suggestions when no solutions could be researched or due insufficient experience. Providing suggestions for these observations is considered future research since we did not have the time to extensively research all observations. Moreover, we aim to provided suggestions that are also applicable to other generator architectures than the one we researched. This is needed to design the reference architecture is a more generic way. Those suggestions are where possible based on literature, experiment results or domain experts from the case study company. Besides the reference architecture itself also documentation is provided in which design decision, protocol and practical notes are described.

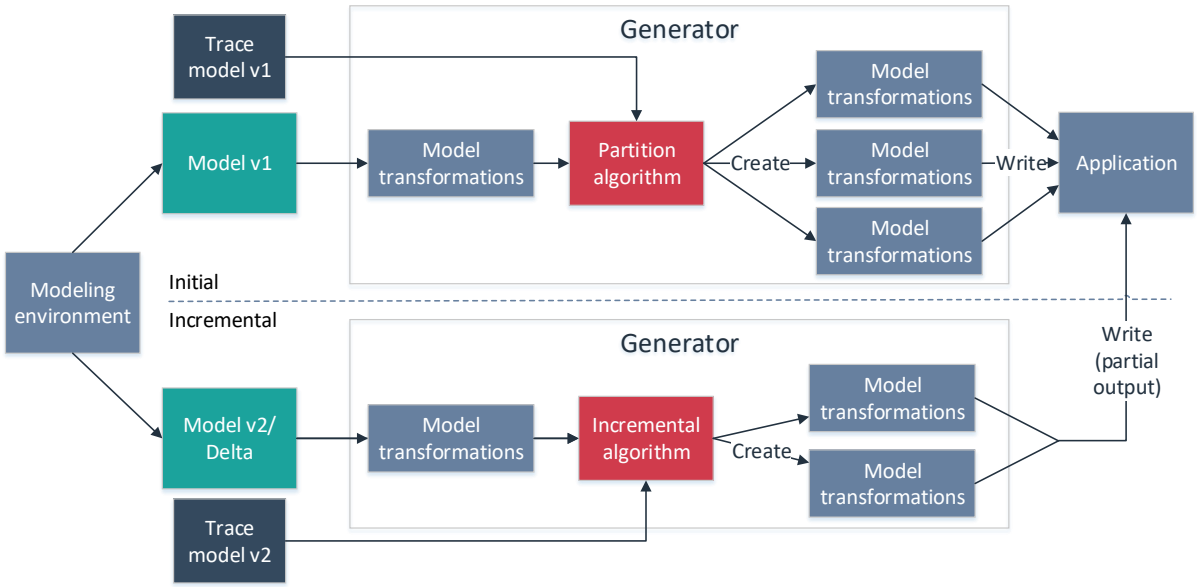

Figure 12: The proposed reference architecture based on identified problems or observations made during the case studies and on solutions to these problems.

During the case studies short-comings and bottlenecks of the current setup are observed. Moreover, we primarily focused on architectural patterns and excluded detailed implementation specifics that might influence the model transformation performance. Only observations on the researched aspects are included and thus not all steps needed to create a MDD generator are explained in detail. Expanding

our proposed reference architecture is left to future research where additional case studies provide new insights and validate existing observations.

In the remainder of this section we gradually built and explain our proposed architecture. All observations and suggestions made are numbered and linked to elements or arrows in the provided figures. First, the initial model transformation scenario is covered and depicted in Figure 13. After that, the incremental generation scenario is covered and shown in Figure 14.

7.1 Initial Model Transformation

7.1.1 Model Partitioning

The first artifact in Model-Driven Development flow are one or more source models (A). These source models contain dependencies and form the traceability model as explained in traceability case study (Section 4). **Observation 5** states that the number of dependencies should be kept to a minimum since dependencies are disadvantageous when creating partitions with an equal distribution of model element types. We propose a solution in which the modeler is made aware of the dependencies he or she created either explicitly or implicitly by modifying the model. A source element relies on a specific property in the target element and thus any change that might modify this property also affects the source element. The modeler could be assisted while creating or modifying the source model about introducing new dependencies and the number of existing dependencies between two elements. It is beneficial to reduce the number of dependencies on properties from other elements. This can be achieved by following the dependency inversion idea of the SOLID programming principles by Robert C. Martin. This means for example that a base element is responsible for the calculation and a dependent element only requests the resulting value. This setup makes sure that a minimal amount of existing artifacts are recreated after a model modification.

> **Solution(s) for Observation 5**: Make the modeler aware of the existing dependencies in the source model. Furthermore, indicate the impact a given model adaption has on the dependency or trace model. SOLID principles can be used to guide the modeler in designing the model that is best structured for model partitioning.

In general, loading the source model(s) into memory is the first operation performed by a MDD generator. This is needed before analysis and thus partitioning of the source models can be performed. We researched this phase and during our parallel generation experiment we observed that additional model transformation time in introduced in the model loading phase. This issue is elaborated in **Observation 12** and corresponds to B in Figure 13. We proposed two solutions to deal with the problem. First, start the pipeline sequentially where is single model is loaded and preparation steps are performed (B). This approach is still equivalent to the data parallelism approach. After that, the partition can be created (C) and executed in parallel in remainder of the pipeline (D). Secondly, in case

there is an existing generation an external tool can be used that partitions the source models. Following this approach the existing generator architecture can be left unaltered and multiple instances of the generator can be executed with each a different model partition. It is the responsibility of the partition algorithm to create partitions that can be generated by itself without additional dependencies to other elements.

Solution(s) for Observation 12:

1. Postpone the model partitioning phase to ensure that only one instance of a source model has to be loaded which is used throughout the remainder of the generator. After the model is loaded it can be analyzed for partitioning and later parallel model transformation.
2. In case an existing generator is used an external tool can be leveraged that partitions the source models.

For both approaches see **Observation 1** and the corresponding solution on issues observed with collecting trace information used for model partitioning.

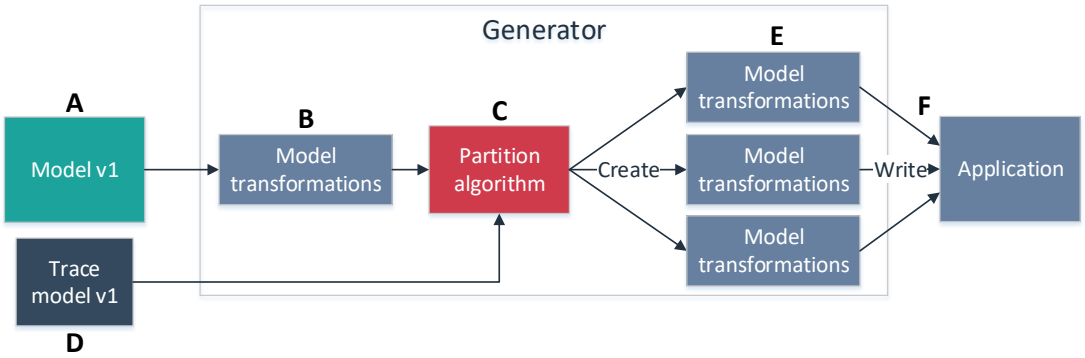

Figure 13: Initial model transformation reference architecture. A source model (A) is passed to the generator. A first set of model transformation, among others, load the model into memory (B). Then a partition algorithm is executed that analyze the in-memory source model (C) and a trace model (D) to partition the model for parallel model transformation (E). The trace model is either based on involved model transformation given a source model or a model of all dependencies in the generator. Finally, each parallel model transformation task write a distinct set of output artifacts that together form the output application.

7.1.2 Compute the Trace Model

Creating the trace model (D) in a generator programmed in a general-purpose language is not straightforward. Moreover, this process is time-consuming since trace generating code is insert by every model transformation as mentioned in **Observation 1**. However, we could not find a solution that is less time-consuming and yields the same result. The traceability approach where trace generating code

is added to model transformation requires that the transformations have to be executed with a source model in order to create a trace model. We consider this as a drawback since a large part of the transformation pipeline if not all is executed to collect all traces as described in **Observation 14**. We propose several solutions to create the trace model more efficiently. First, we describe a simulation approach where only a part of the transformation pipeline is executed up to a certain point. At this point all dependencies that exist are either known or can be deduced. This approach might require changes in the architecture to ensure that at some point only one-to-one transformation are executed of which the dependencies can be deduced. Since no new dependencies are introduced beyond this point there is no need to generate any further. It is of course desirable that this point comes as early in the transformation pipeline as possible. A re-run of the pipeline is needed with the new source model and the newly created trace model to generate the output artifacts themselves. Secondly, we propose an alternative method where model transformations are formalized to enable static analysis and thereby collecting trace information. A downside of this approach is that the abstract and imperative transformation rules must be kept in sync. A second issue is that all transformations must be extracted and rewritten in a formal language. A benefit of this approach is that the trace information is independent from source models and can be reused for all end user. Lastly, theory from abstract interpretation could be used to statically analyze the model transformations. Muthukumar and Hermenegildo (1992) propose a new algorithm to analyze logic programs by using abstract interpretation. The algorithm focus on inferring dependencies between program expressions. However, no direct applicable approach is found for transformations programmed in a general-purpose language.

Solution(s) for Observation 14:

1. A simulation approach where a part of the transformation pipeline is executed with the new model to gather traceability information. The pipeline is re-run to utilize the trace model for model partitioning to generate in parallel.
2. Formalize all model transformation and store them as a separate artifact. The formalization ensures that static analysis can be applied to create the trace info. The created trace model can be reused for all end users that use that version of the generator.
3. Use abstract interpretation to statically collect dependencies in model transformations.

7.1.3 Apply the Trace Model

In **Observation 10** we observed that partition code was duplicated and used at multiple locations in the transformation pipeline. Before we already suggested a partition algorithm and therefore we also include this as a module in our reference architecture (C) to solve the observation. This module contains all partitioning

logic in a central location and use the generated traceability model to partition the source models into a number of partitions. These partitions should be complete to make sure no external dependencies are present. These partitions are then generated in parallel (E).

> **Solution(s) for Observation 10**: We propose a model partitioning module which should be a single phase in the generation pipeline that partitions the source model(s). By conforming to this property we can also provide a solution to **Observation 12**, where we observed that the entire model is needed for each parallel instance. The created partitions are then given to the parallel generation instances to parallel model transformation.

We argue that the most important feature of an efficient partitioning algorithm is that small and evenly distributed partitions are created. This can be achieved by selecting the right model element type with the right granularity. This is easier said than done since no suitable element type or unit is found that is best used to do model partitioning. Furthermore, no element properties could be defined taht are needed to be effective and efficient for partitioning and is elaborated in **Observation 7**.

7.1.4 Practical Notes

During our case study we researched how to best implement our partial generation algorithm. No literature was found and resulted in **Observation 9**. However, we could not provide any well-reasoned suggestions for the observation and is left to future research. Moreover, compiling the output artifacts separately did not work in D as observed in the partial generation case study and is elaborated in **Observation 8**. We describe a work-around on how to deal with this observed issue. The work-around is to avoid generating source code for a compiled language. This is the approach our case study organization is already migrating to. This approach entails that the generator generates artifacts that are interpreted at runtime.

> **Solution(s) for Observation 8**: Avoid generating source code for a compiled language in partial generation. This eliminates many dependencies between output artifacts. Instead generate artifacts that can be interpreted. Since the actual use of the artifacts are delayed to runtime all partial outputs are combined which means that all dependencies can be resolved.

Lastly, the partial model transformation tasks at D (Figure 13) all write their output to a single output location (F). In **Observation 11** we described the file locking conflict problem with this approach. Therefore, we propose a solution by ensuring, where possible, that the partial model transformation tasks to not produce overlapping output artifacts. This eliminates the need to copy different

output folders and to merge files with the duplicate filenames. The internals of the generator must be adapted in such a way that each partial task creates a disjoint set of artifacts. This can primarily be achieved by conforming to this requirement during model partitioning.

> **Solution(s) for Observation 11**: Ensure that the partial model transformation tasks to not produce overlapping output artifacts. This eliminates the need to copy different output folders and to merge files.

7.2 Incremental Model Transformation

The incremental model transformation scenario starts somewhat different compared to the initial scenario. There are two versions of the source model, namely the previous model which is already transformed into an application, and the newly created model. Model modifications are made incrementally in the same way that an application is incrementally updated according to the model changes. The two model versions are then passed to the model transformation phase F in Figure 14). In **Observation 13** we mentioned the inefficiency of an approach where the generator determined the delta given the two source models. We argue that it might not be needed to load two complete models to determine the delta. In case the developers have control over the modeling environment it would be beneficial to keep track of changes whilst they are made. This way the delta itself is directly created and does not have to be computed using the two model versions. We see from the experiments that it takes 0.5 second to load a small model (less than 30 elements), like a delta, whereas it takes about 20 seconds to load the largest tested model (more than 1000 elements). Moreover, the modeling environment has knowledge and access to the latest model version and can thus directly include other elements that are connected through dependency relationships. This of course works only for dependencies in the model and traceability for model transformations is still required. However, we were unable to propose a solution for situations where no access to the modeling tool is available.

> **Solution(s) for Observation 13**: We propose a solution where only the delta model itself is passed the generator. Access is needed to the modeling tool to directly create a delta model when a change is made by the end user.

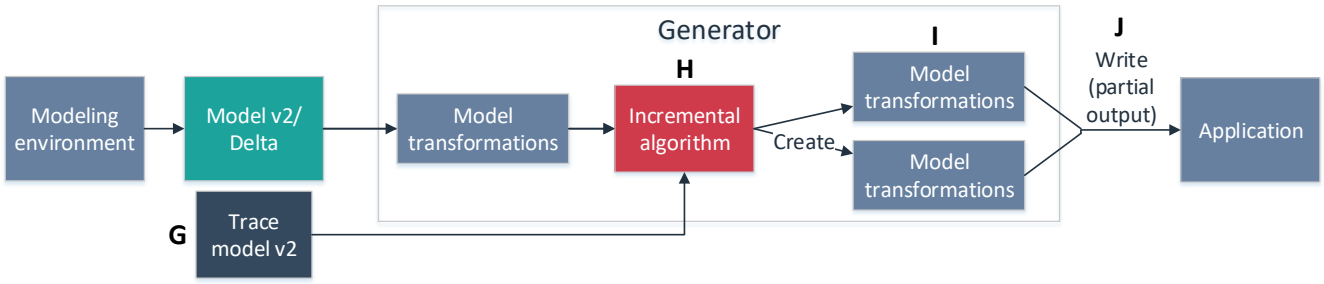

Figure 14: Reference model evolution handling after the initial model transformation. An updated source model is passed to the generator which is loaded into memory. After that an updated version of the trace model must be calculated (G) on which the Incremental algorithm is dependent (H). Then the calculated delta model can be transformed by the generated incrementally and in possibly parallel depending on the overhead (I). Finally, a partial output is written to the already existing set of application files to update the application.

After the delta is loaded it must be transformed to update the application. Before this process can start additional calculations on the delta are needed to complement the delta with generation-time dependency elements. This is explained in detail in the Related work and Traceability case study sections. We group the logic required for these calculations in a new and separate module and place it into the model transformation pipeline. We call this module "Incremental algorithm" and can be found at letter I in Figure 14. This module requires a delta model and a traceability model. A traceability model is created in the same way as for the initial model transformation described earlier where it is used for model partitioning. For the simulation approach the new source model is used since the delta cannot be used to create a complete trace model.

Again, the last phase of the generator is to write the created artifacts to the application folder (J in Figure 14). Since at this point only a partial output is created the new and existing artifacts have to be merged in some way. As already described earlier file locking conflicts could occur when naively merging the partial out with existing artifacts. This problem can be solved by the suggestion provided for **Observation 11** above.

7.3 Open Concerns Regarding Traceability

While researching traceability we made additional observations that do not influence the design of the reference architecture. These observations are of a more practical nature and are therefore also important to take into account. First, inserted trace generation code must be kept up-to-date otherwise it cannot be used reliably (**Observation 2**). An IDE that can support the developer in this task but we could not find such an IDE or any other mechanism to aid the developer in keeping the code snippets up-to-date. **Observation 3** argues that only relevant traces have to be collected. We consider it the responsibility of the developer to

determine what is relevant what is not. However, no structured protocol or guidelines were found that could aid the developer in this process. This can reduce the number of traces collected which increase model transformation performance as is discussed in **Observation 6**. However, no structured method to do this is found in literature because of the many implementation details and possibly language specifics. Lastly, a suitable format in which the trace model can be stored is researched and analyzed in **Observation 4**. We could not compare both XMI and JSON formats in our case study generator and can therefore not provide a best practice or solution and depends also on the transformation platform used.

8 Discussion and Opportunities

8.1 Findings and Implications

The most important finding in our research is the complexity of traceability in a MDD generator written in a general-purpose language. Inserting trace generating code into model transformations is time-consuming. However, a more efficient method is not proposed in the MDD literature. As a result, no complete traceability model could be generated. This in turn means that parallel generation could only be researched on a basic level and incremental generation could not be researched as part of a case study at all. All findings and corresponding implications are described as observations and solutions in Section 7.

8.2 Limitations

This research is part of a master thesis which means time was a large constraint. This constraint limits our research in several ways. First, only a limited number of approaches to improve the performance could be researched and validated. Secondly, a single case study organization could be researched which leaves the reference architecture novel and unvalidated. Limitations in applicable available literature changed our research direction to a more exploratory research and could only address certain topics briefly and in high level. The experiment results are only limited generalizable since the time for creating partitions is not accounted for in the total model transformation time. Moreover, no experiment could be performed to prove a performance improvement of incremental model transformation.

8.3 Opportunities

Many limitations are recognized in the conducted research, therefore, several interesting opportunities can be identified for further research. The primary opportunity to actually improve the performance of model transformation is to find and use an efficient approach to built a trace model. A requirement is that the trace model creation process must be automated and is preferably more feasible than inserting trace generating code snippets which has additional drawbacks attached to it as explained in Section 7.3. We briefly looked into abstract interpretation which has potential to quickly determine all dependencies independent of any source model. Statically analyzing the dependencies of a Prolog program is already researched. Therefore, extending their methods to handle imperative languages is interesting and considered future research.

9 Conclusion

This section concludes the research by answering the designed sub-research questions. Finally, the main research question is answered.

SRQ1 *What is the current state of methods that reduce generation time in Model-Driven Development?*

A related work study and analysis is performed to explore the available methods for improving the performance of a MDD generator. The domain of Model-Driven Development exist already for a long time and have matured over the years. Multiple transformation languages and tools are proposed where ATL is the most mature. All researched approaches aimed at improving the performance use ATL as the transformation language and leverage the underlying virtual machine for optimizations. The extensive focus on ATL is a drawback since the case organization use a different transformation language, i.e. a general-purpose programming language. This has significantly different properties compared to ATL that is a Domain Specific Language regarding model transformation. There is only little applicable literature available to our context which limits this research in the number of approaches that can be researched.

SRQ2 *How is insight gained in the model transformations in a Model-Driven Development environment?*

A common approach to gain insight into the transformations of a MDD generator is traceability. Traces are collected from the model transformations in the MDD generator and form a trace model. Traces are dependencies between two elements that take part in a model transformation. By analyzing the trace model the MDD generator can be visualized in terms of model dependencies. Moreover, it is used to reason about how a model is transformed into an output application. We consider traceability as a prerequisite for both model partitioning and incremental model transformation. Our case study shows that it is complex to collect all traces needed to use it effectively for model partitioning of incremental model transformation. We observed that it is time-consuming to modify our case generator due to the general-purpose language used for the model transformations. Since this approach is very expensive or even unfeasible another approach is briefly researched, namely abstract interpretation.

SRQ3 *How can these insights be utilized to improve the model transformation performance in a Model-Driven Development environment?*

The insights provided by the trace model are utilized in the generator by partitioning the source model into one or more partitions. This can be achieved by analyzing the dependencies captured in the trace model and enables partial - and incremental model transformation. We researched a data parallelism approach which provides all components needed for partial model transformation.

A valid partition can only be created when the trace model contains all possible dependencies given a source model. Valid here means that the partition can be generated without additional knowledge of other elements or dependencies. By implementing the data parallelism approach as part of a case study observations could be made which provide insights on what properties and processes are needed to improve transformation performance by using partial model transformation.

SRQ4 *Does the utilization of the insights lead to an improved performance?*

Finally, parallel and incremental model transformation are researched in a case study. Multiple partial model transformation tasks are combined to research parallel model transformation. An experiment is performed to prove that the implemented parallelization technique does indeed work and yield a performance improvement compared to sequential model transformation. Our parallel transformation reduce the total generation duration by 2.5 to 3.5 times which corresponds to experiment performed with ATL on MapReduce. A limitation of our approach is that the model partitioning is not based on traces since no complete trace model could be created. Instead, a certain model element type is used and the explicit dependencies from the source model are used. This approach does not work for a source model with more complex relationship where dependency information from the model transformations are needed to be able to generate a correct output.

RQ *How can the model transformation time be reduced in Model-Driven Development?*

The main research question is answered by our presented reference architecture and the corresponding design decision, protocols and practical take-aways described in Section 7. The reference architecture and corresponding documentation combined present a novel artifact and is designed to aid organizations in improving their MDD generator. We proved and argued that both parallel - and incremental model transformation, respectively, improve the performance of model transformations in MDD. The reference architecture is based on observations made during the case studies and single experiment and are to be extended and validated by performing additional case studies.

References

Aizenbud-Reshef, N., Nolan, B. T., Rubin, J., and Shaham-Gafni, Y. (2006).
Model traceability.
IBM Systems Journal, 45(3):515–526.

Benelallam, A., Gómez, A., and Tisi, M. (2015a).
ATL-MR: Model Transformation on MapReduce.
Proceedings of the 2nd International Workshop on Software Engineering for Parallel Systems, pages 45–49.

Benelallam, A., Gómez, A., Tisi, M., and Cabot, J. (2015b).
Distributed Model-to-Model Transformation with ATL on MapReduce.
Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering, pages 37–48.

Benelallam, A., Tisi, M., Sánchez Cuadrado, J., De Lara, J., and Cabot Ef, J. (2016).
Efficient Model Partitioning for Distributed Model Transformations- ficient Model Partitioning for Distributed Model Transformations.
ACM SIGPLAN.

Bergmann, G., Horváth, Á., Ráth, I., Varró, D., Balogh, A., Balogh, Z., & Ökrös, A. (2010).
Model Driven Engineering Languages and Systems.
Model Driven Engineering Languages and Systems, pages 76–90.

Brown, A. (2004).
An introduction to Model Driven Architecture Part I: MDA and today's systems.
IBM DeveloperWorks, RationalEdge.

Clasen, C., Didonet, M., Fabro, D., and Tisi, M. (2012).
Transforming Very Large Models in the Cloud: a Research Roadmap.
In *First International Workshop on Model-Driven Engineering on and for the Cloud*. Springer.

Czarnecki, K. and Helsen, S. (2003).
Model-Driven Architecture Classification of Model Transformation Approaches.
OOPSLA'03 Workshop on Generative Techniques.

Czarnecki, K. and Helsen, S. (2006).
Feature-based survey of model transformation approaches &.
IBM Systems Journal, 45(3):621–645.

Dean, J. and Ghemawat, S. (2008).
MapReduce: Simplified Data Processing on Large Clusters.
Communications of the ACM, 51(1):–.

Dongarra, J. J. and Sorensen, D. C. (1987).
A portable environment for developing parallel FORTRAN programs *.
Parallel Computing, 5:175–186.

France, R. and Rumpe, B. (2007).
Model-Driven Development of Complex Software: A Research Roadmap.
Workshop on the Future of Software Engineering (FOSE 2007), at the 29th International Conference on Software Engineering (ICSE 2007), Minneapolis, Minnesota, USA, (May 2007):37–54.

Hailpern, B. and Tarr, P. (2006).
Model-driven development: The good, the bad, and the ugly.
IBM SYSTEMS JOURNAL, 45(3).

Hearnden, D., Lawley, M., and Raymond, K. (2006).
Incremental Model Transformation for the Evolution of Model-Driven Systems.
In *Model Driven Engineering Languages and Systems, 9th International Conference, MoDELS 2006, Genova, Italy, October 1-6, 2006, Proceedings*, volume 4199, pages 321–335. Springer Berlin Heidelberg.

Hevner, A. R., March, S. T., Park, J., and Ram, S. (2004).
DESIGN SCIENCE IN INFORMATION SYSTEMS RESEARCH 1.
Design Science in IS Research MIS Quarterly, 28(1):75–105.

Huchard, M., Nebut, C., Transformations, M., Neple, T., and Ecmda, J. O. (2006).
Towards a Traceability Framework for Model Transformations in Kermeta Towards a traceability framework for model.

Hussmann, H., Meixner, G., and Zuehlke, D. (2011).
Model-Driven Development of Advanced User Interfaces.

Inostroza, P., van der Storm, T., and Erdweg, S. (2014).
Tracing Program Transformations with String Origins.
In *International Conference on Theory and Practice of Model Transformations*, pages 154–169.

Jouault, F. (2005).
Loosely Coupled Traceability for ATL.
In *Proceedings of the European Conference on Model Driven Architecture (ECMDA) workshop on traceability*, volume 91.

Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I., and Valduriez, P. (2006).
ATL : a QVT - like Transformation Language.
Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications, pages 719–720.

Jouault, F. and Tisi, M. (2011).
Towards Incremental Execution of ATL Transformations.
ICMT, 10:123–137.

Kabbedijk, J., Jansen, S., and Brinkkemper, S. (2012).
A case study of the variability consequences of the CQRS pattern in online business software.
In *Proceedings of the 17th European Conference on Pattern Languages of Programs - EuroPLoP '12*, pages 1–10, New York, New York, USA. ACM Press.

Lussenburg, V., van der Storm, T., Vinju, J., and Warmer, J. (2010).
Mod4J: a qualitative case study of model-driven software development.
Model Driven Engineering Languages and Systems, pages 346–360.

Muthukumar, K. and Hermenegildo, M. (1992).
Compile-time derivation of variable dependency using abstract interpretation.
The Journal of Logic Programming, 13(2-3):315–347.

Olsen, G. K. and Oldevik, J. (2007).
Scenarios of Traceability in Model to Text Transformations.
In *Model Driven Architecture- Foundations and Applications*, pages 144–156. Springer Berlin Heidelberg, Berlin, Heidelberg.

Polya, G. (2014).
How to solve it: A new aspect of mathematical method.

Schunselaar, D. M., Gulden, J., van der Schuur, H., and Reijers, H. A. (2016).
A Systematic Evaluation of Enterprise Modelling Approaches on Their Applicability to Automatically Generate Software.
In *Business Informatics (CBI), 2016 IEEE 18th Conference on*, pages 290–299.

Selic, B. (2003).
The pragmatics of Model-Driven Development.
IEEE Software, 20(5):15–25.

Sendall, S. and Kozaczynski, W. (2003).
Model Transformation – the Heart and Soul of Model-Driven Software Development.

Staron, M., Kuzniarz, L., and Wallin, L. (1994).
Nordic journal of computing., volume 11.
Pub. Association Nordic Journal of Computing.

Subhlok, J., Stichnoth, J. M., O 'hallaron, D. R., and Gross, T. (1993).
Exploiting Task and Data Parallelism on a Multicomputer.
Fourth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming.

Tisi, M., Martinez, S., and Choura, H. (2013).
Parallel Execution of ATL Transformation Rules.
Parallel Execution of ATL Transformation Rules. MoDELS, pages 656–672.

Tisi, M., Martínez, S., Jouault, F., and Cabot, J. (2011).
Lazy Execution of Model-to-Model Transformations.
pages 32–46. Springer Berlin Heidelberg.

Trask, B., Paniscotti, D., Roman, A., and Bhanot, V. (2006).
Using model-driven engineering to complement software product line engineering in developing software defined radio components and applications.
In *Companion to the 21st ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications - OOPSLA '06*, page 846, New York, New York, USA. ACM Press.

van Kooten, J. (2016).
Do model-driven development (mdd) platforms make good on their promises?
June 19, 2017, Retrieved from http://bpmcompany.eu/blog/109-do-model-driven-development-mdd-platforms-make-good-on-their-promises.

Varró, D. (2015).
Patterns and styles for incremental model transformations.
PAME@ STAF, pages 41–43.

Weigert, T. and Weil, F. (2006).
Scholars' Mine Practical Experiences in using Model-Driven Engineering to Develop Trustworthy Computing Systems Practical Experiences in Using Model-Driven Engineering to Develop Trustworthy Computing Systems.
Computer Science Faculty Research.

Zima, H. P., Bast, H.-J., and Gerndt, M. (1988).
SUPERB: A tool for semi-automatic MIMD/SIMD parallelization *.
Parallel Computing, 6:1–18.

Appendices

A Experiment models specified per element type

Table 2: Model for test 1 with 5 parallel instances.

Test case	# Entities	# Roles	# Events	# Work areas	Total
1	5	5	90	4	104
2	10	10	180	4	204
3	15	15	270	4	304
4	20	20	360	4	404
5	25	25	450	4	504
6	30	30	540	4	604
7	35	35	630	4	704
8	40	40	720	4	804
9	45	45	810	4	904
10	50	50	900	4	1004

Table 3: Model for test 2 with 6 parallel instances.

Test case	# Entities	# Roles	# Events	# Work areas	Total
1	5	5	90	5	105
2	10	10	180	5	205
3	15	15	270	5	305
4	20	20	360	5	405
5	25	25	450	5	505
6	30	30	540	5	605
7	35	35	630	5	705
8	40	40	720	5	805
9	45	45	810	5	905
10	50	50	900	5	1005

Table 4: Model for test 3 with 7 parallel instances.

Test case	# Entities	# Roles	# Events	# Work areas	Total
1	5	5	90	6	106
2	10	10	180	6	206
3	15	15	270	6	306
4	20	20	360	6	406
5	25	25	450	6	506
6	30	30	540	6	606
7	35	35	630	6	706
8	40	40	720	6	806
9	45	45	810	6	906
10	50	50	900	6	1006

Table 5: Model for test 4 with 8 parallel instances.

Test case	# Entities	# Roles	# Events	# Work areas	Total
1	5	5	90	7	107
2	10	10	180	7	207
3	15	15	270	7	307
4	20	20	360	7	407
5	25	25	450	7	507
6	30	30	540	7	607
7	35	35	630	7	707
8	40	40	720	7	807
9	45	45	810	7	907
10	50	50	900	7	1007

Table 6: Model for test 5 with 9 parallel instances.

Test case	# Entities	# Roles	# Events	# Work areas	Total
1	5	5	90	8	108
2	10	10	180	8	208
3	15	15	270	8	308
4	20	20	360	8	408
5	25	25	450	8	508
6	30	30	540	8	608
7	35	35	630	8	708
8	40	40	720	8	808
9	45	45	810	8	908
10	50	50	900	8	1008

Table 7: Model for test 6 with 10 parallel instances.

Test case	# Entities	# Roles	# Events	# Work areas	Total
1	5	5	90	9	109
2	10	10	180	9	209
3	15	15	270	9	309
4	20	20	360	9	409
5	25	25	450	9	509
6	30	30	540	9	609
7	35	35	630	9	709
8	40	40	720	9	809
9	45	45	810	9	909
10	50	50	900	9	1009

Table 8: Model for test 7 with 11 parallel instances.

Test case	# Entities	# Roles	# Events	# Work areas	Total
1	5	5	90	10	110
2	10	10	180	10	210
3	15	15	270	10	310
4	20	20	360	10	410
5	25	25	450	10	510
6	30	30	540	10	610
7	35	35	630	10	710
8	40	40	720	10	810
9	45	45	810	10	910
10	50	50	900	10	1010

Table 9: Model for test 8 with 5 parallel instances and fewer model elements.

Test case	# Entities	# Roles	# Events	# Work areas	Total
1	10	0	12	4	27
2	10	0	24	4	39
3	10	0	36	4	51
4	10	0	48	4	63
5	10	0	60	4	75
6	10	0	72	4	87
7	10	0	84	4	99
8	10	0	96	4	111
9	10	0	108	4	123
10	10	0	120	4	135

Table 10: Model for test 9 with 4 parallel instances and fewer model elements.

Test case	# Entities	# Roles	# Events	# Work areas	Total
1	10	0	12	3	26
2	10	0	24	3	38
3	10	0	36	3	50
4	10	0	48	3	62
5	10	0	60	3	74
6	10	0	72	3	86
7	10	0	84	3	98
8	10	0	96	3	110
9	10	0	108	3	122
10	10	0	120	3	134

Table 11: Model for test 10 with 3 parallel instances and fewer model elements.

Test case	# Entities	# Roles	# Events	# Work areas	Total
1	10	0	12	2	25
2	10	0	24	2	37
3	10	0	36	2	49
4	10	0	48	2	61
5	10	0	60	2	73
6	10	0	72	2	85
7	10	0	84	2	97
8	10	0	96	2	109
9	10	0	108	2	121
10	10	0	120	2	133

B Experiment results

B.1 Results of 5 parallel generation instances

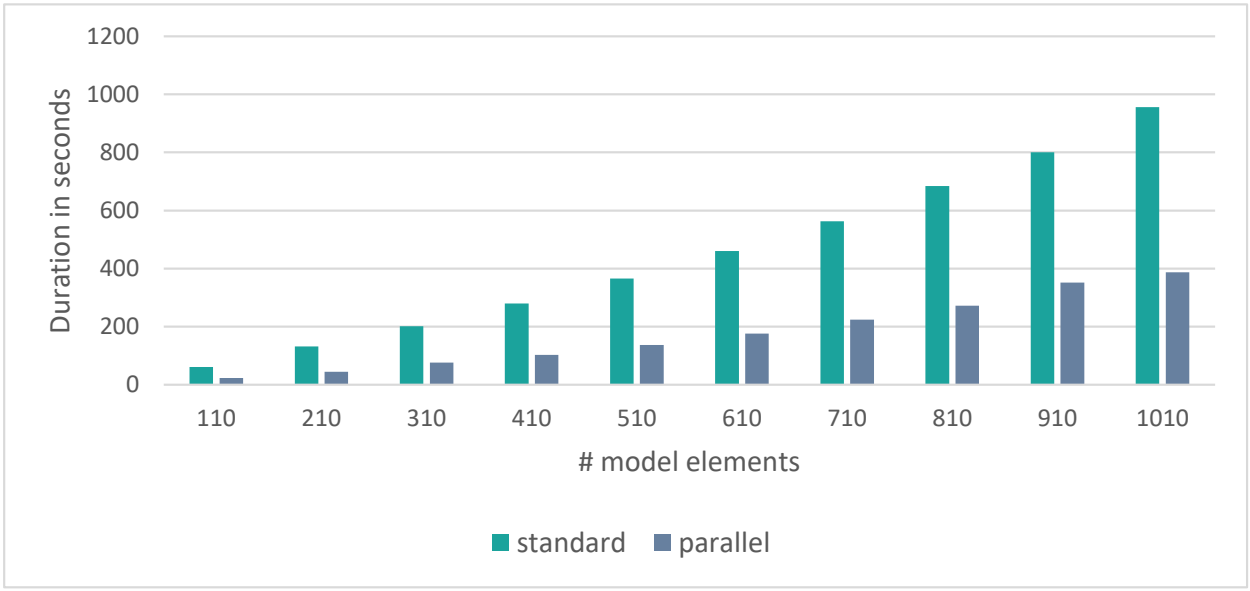

Figure 15: Generation duration chart with standard model generation compared to generation with 5 parallel generation instances.

Table 12: Test 1: baseline test.

# elem. / Run	110	210	310	410	510	610	710	810	910	1010
run 1	61,355	136,009	199,128	279,472	364,739	457,927	563,404	674,544	824,683	960,786
run 2	61,294	131,802	201,895	279,147	365,229	461,461	564,311	698,512	815,380	964,431
run 3	62,228	127,037	205,158	280,005	366,431	458,954	562,740	674,380	822,387	956,996
run 4	61,542	130,760	201,709	279,066	365,603	458,106	571,613	674,636	825,104	969,137
run 5	61,737	129,057	203,893	284,095	366,593	460,920	560,247	671,022	830,898	959,646
run 6	61,464	140,212	203,270	279,429	367,714	459,029	562,401	672,536	828,220	963,376
run 7	61,816	129,122	201,473	279,039	366,888	459,926	562,521	675,545	825,998	968,039
run 8	61,478	126,685	202,885	279,262	366,363	461,001	563,896	671,286	819,555	962,220
run 9	61,556	131,311	200,543	278,924	364,855	469,690	561,585	711,485	823,255	970,980
run 10	61,605	130,260	199,433	279,314	365,060	458,429	562,002	719,938	818,346	969,923
average	61,608	131,226	201,939	279,775	365,948	460,544	563,472	684,388	823,383	964,553
SD	0,269	4,115	4,115	1,922	0,995	3,453	3,089	18,435	4,672	4,779

Table 13: Test 1: 5 parallel tasks.

# elem. / Run	110	210	310	410	510	610	710	810	910	1010
run 1	22,156	43,004	72,538	105,935	137,065	177,768	224,530	278,212	332,887	388,781
run 2	23,342	49,966	69,846	102,116	135,543	174,306	222,217	271,185	329,790	390,742
run 3	23,963	43,218	80,830	102,339	137,737	174,791	223,147	274,842	327,475	388,197
run 4	22,662	44,062	80,780	103,187	136,509	176,546	221,901	275,947	330,897	390,540
run 5	23,367	43,318	70,683	102,259	132,922	177,453	223,595	272,204	325,946	388,655
run 6	23,411	43,715	70,808	107,315	141,775	177,483	225,802	275,015	330,065	389,093
run 7	22,675	43,632	79,770	101,016	135,543	176,777	227,561	268,903	333,018	389,922
run 8	23,488	43,977	70,915	100,844	133,807	179,201	224,653	273,124	331,364	390,493
run 9	24,193	43,605	73,732	101,443	137,150	177,120	225,696	267,472	330,156	390,148
run 10	23,957	44,696	86,825	101,067	133,114	175,418	223,835	267,839	336,421	388,806
average	23,321	44,319	75,673	102,752	136,117	176,686	224,294	272,474	330,802	389,538
SD	0,652	2,041	2,041	5,893	2,624	1,483	1,737	3,628	2,940	0,929

B.2 Results of 6 parallel generation instances

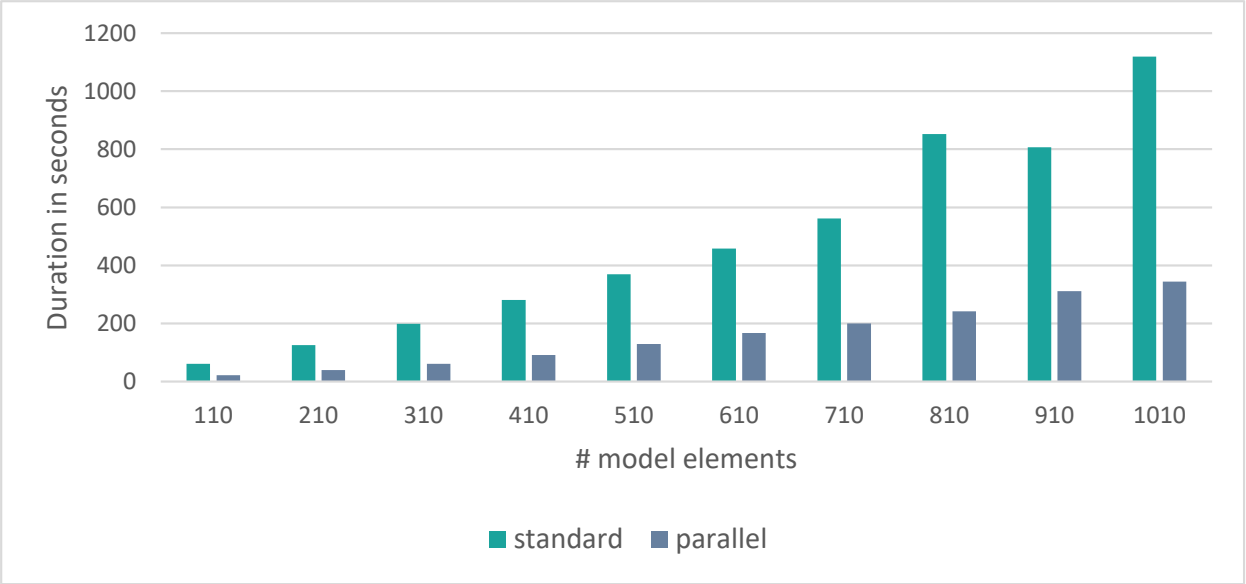

Figure 16: Generation duration chart with standard model generation compared to generation with 6 parallel generation instances.

Table 14: Test 2: baseline test.

# elem. / Run	110	210	310	410	510	610	710	810	910	1010
run 1	61,219	123,778	196,221	274,878	360,397	457,911	560,653	669,879	794,160	937,534
run 2	60,868	123,453	195,171	275,779	359,797	459,750	562,445	667,939	793,618	948,904
run 3	60,728	122,706	195,099	274,345	360,543	458,100	561,641	668,366	792,248	935,347
run 4	61,224	123,661	195,498	274,704	360,815	458,018	561,109	668,297	827,855	939,818
run 5	60,997	123,108	195,876	275,099	359,913	459,160	563,946	666,823	800,797	945,793
run 6	60,340	123,487	195,578	275,082	360,946	459,992	563,697	669,739	810,070	1021,749
run 7	60,833	123,597	195,236	275,524	360,847	458,258	562,966	670,719	805,858	1110,181
run 8	60,990	123,377	196,462	274,785	359,565	458,365	561,733	668,008	809,632	1100,623
run 9	60,711	122,788	195,812	273,596	358,765	458,723	560,458	668,275	797,255	1051,533
run 10	60,907	123,153	195,320	274,864	360,510	458,343	559,500	668,214	807,814	952,525
average	60,882	123,311	195,627	274,866	360,210	458,662	561,815	668,626	803,931	994,401
SD	0,259	0,363	0,363	0,459	0,692	0,734	1,449	1,143	10,822	70,385

Table 15: Test 2: 6 parallel tasks.

Run \ # elem.	110	210	310	410	510	610	710	810	910	1010
run 1	21,467	41,413	63,526	91,756	124,024	160,357	204,696	245,745	292,093	343,462
run 2	23,758	39,997	61,749	90,903	122,649	164,352	202,323	248,066	293,452	341,151
run 3	24,046	40,055	62,464	92,689	124,484	158,476	203,557	248,516	292,342	338,699
run 4	22,071	40,678	63,265	90,961	125,811	158,658	199,483	245,680	288,250	339,552
run 5	23,415	40,366	64,155	94,016	126,774	160,336	206,922	246,997	288,817	340,263
run 6	24,466	39,624	60,973	93,141	124,716	173,442	203,230	247,050	291,381	339,419
run 7	22,538	40,575	63,335	91,919	125,874	163,006	202,344	251,946	288,903	342,540
run 8	23,518	40,418	63,253	92,117	122,653	162,998	204,725	247,308	288,551	341,813
run 9	24,064	40,251	62,702	91,148	125,443	164,302	207,819	247,272	291,502	342,941
run 10	24,062	40,302	63,511	90,948	125,653	160,996	205,616	244,071	291,555	340,946
average	23,341	40,368	62,893	91,960	124,808	162,692	204,072	247,265	290,685	341,079
SD	0,988	0,476	0,476	0,946	1,381	4,334	2,437	2,089	1,869	1,608

B.3 Results of 7 parallel generation instances

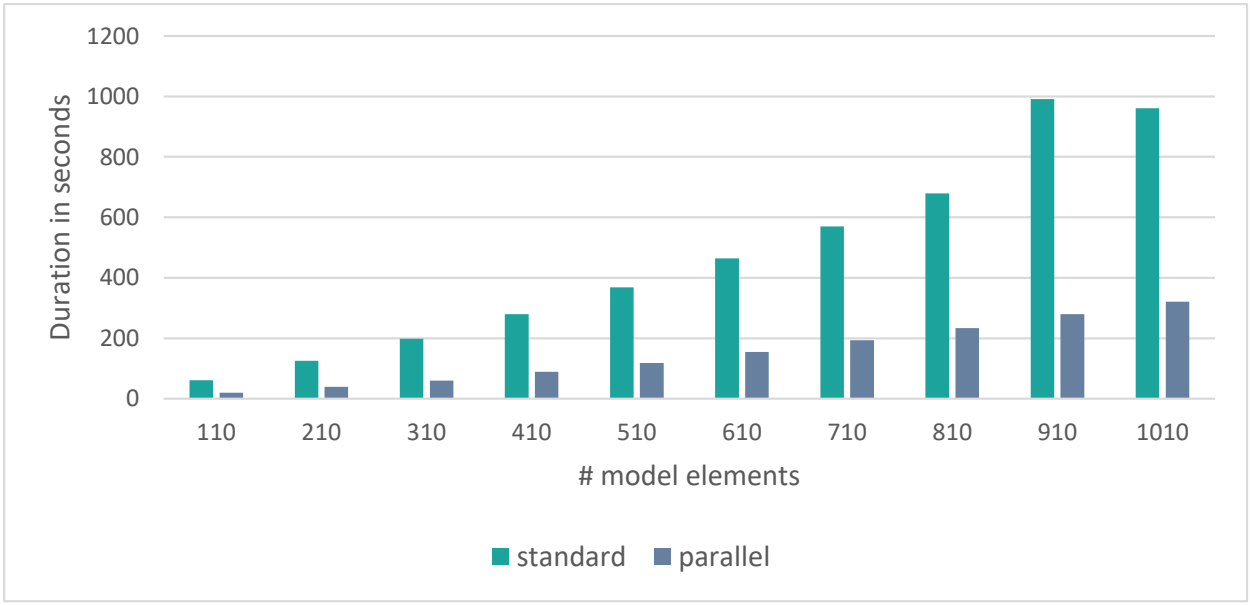

Figure 17: Generation duration chart with standard model generation compared to generation with 7 parallel generation instances.

Table 16: Test 3: baseline test.

# elem. / Run	110	210	310	410	510	610	710	810	910	1010
run 1	61,662	124,901	198,243	281,015	367,636	462,121	569,593	682,237	895,989	956,820
run 2	61,395	125,409	198,856	279,839	369,977	463,154	569,423	677,809	806,782	979,169
run 3	61,814	125,334	199,661	278,182	366,876	462,684	569,129	678,009	850,469	959,913
run 4	61,376	125,029	198,859	280,793	367,987	464,754	570,105	680,334	971,261	964,296
run 5	61,597	125,473	198,186	279,308	366,905	465,576	569,853	680,195	1022,688	963,072
run 6	61,091	124,529	198,810	278,956	366,881	464,367	569,139	679,282	1115,180	958,789
run 7	61,763	125,099	199,466	278,917	368,474	463,128	569,826	679,250	901,585	956,709
run 8	61,678	124,577	198,324	277,212	366,387	464,338	568,844	679,685	1168,466	958,606
run 9	61,807	125,433	197,827	279,138	370,183	463,321	569,002	677,671	1235,839	957,838
run 10	61,885	125,002	199,094	278,346	368,811	463,710	567,846	680,386	940,248	956,913
average	61,607	125,079	198,733	279,171	368,012	463,715	569,276	679,486	990,851	961,213
SD	0,249	0,341	0,341	0,586	1,334	1,042	0,649	1,415	142,024	6,825

Table 17: Test 3: 7 parallel tasks.

# elem. / Run	110	210	310	410	510	610	710	810	910	1010
run 1	22,258	38,928	60,688	88,472	117,498	154,318	195,201	234,046	275,321	323,580
run 2	20,407	38,676	59,546	87,847	118,406	154,538	195,097	234,242	337,288	315,418
run 3	20,381	39,786	60,058	89,916	120,912	152,496	194,286	229,584	273,446	323,989
run 4	20,541	39,693	60,637	88,229	119,395	155,088	190,528	232,489	272,408	321,514
run 5	18,814	37,177	59,680	85,938	118,766	156,423	191,840	235,764	274,788	325,786
run 6	20,738	39,493	59,758	90,523	118,729	154,728	195,295	233,558	273,993	319,721
run 7	18,998	38,709	58,812	87,753	116,492	151,102	192,157	232,772	273,661	320,156
run 8	20,328	37,034	60,627	87,098	117,967	157,727	191,463	234,495	269,957	320,935
run 9	18,932	38,714	60,615	90,854	119,833	154,900	190,286	233,911	270,403	321,670
run 10	20,655	39,566	57,591	87,200	119,782	155,169	194,056	236,126	272,413	321,131
average	20,205	38,778	59,801	88,383	118,778	154,649	193,021	233,699	279,368	321,390
SD	1,050	0,979	0,979	0,991	1,279	1,841	1,977	1,842	20,424	2,813

B.4 Results of 8 parallel generation instances

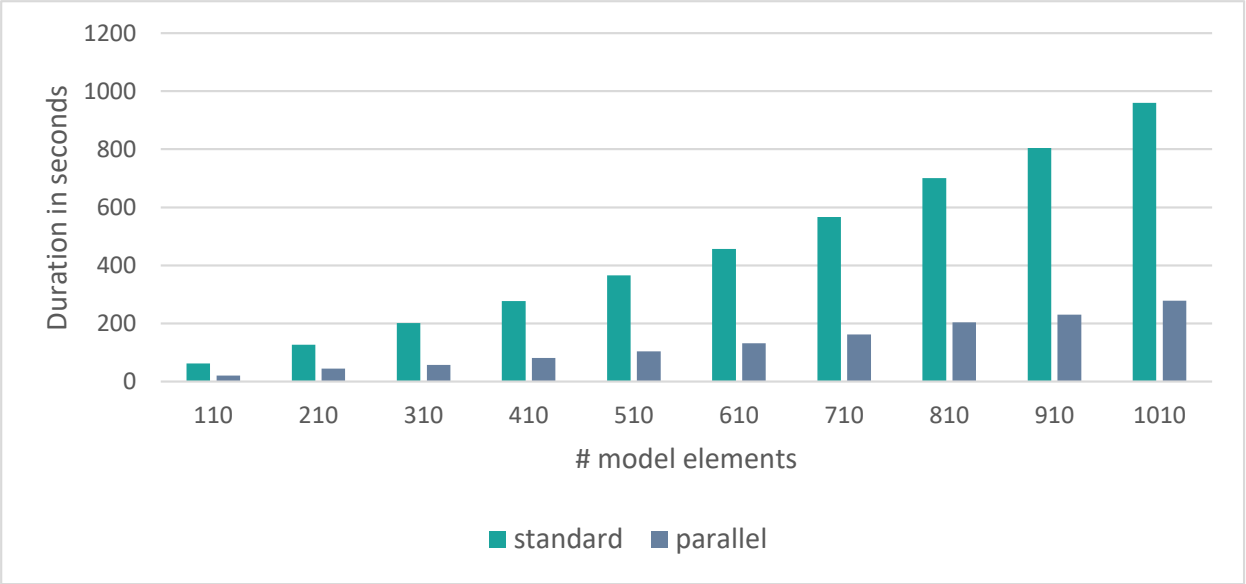

Figure 18: Generation duration chart with standard model generation compared to generation with 8 parallel generation instances.

Table 18: Test 4: baseline test.

# elem. / Run	110	210	310	410	510	610	710	810	910	1010
run 1	62,349	126,949	202,270	285,049	376,758	474,837	577,885	691,472	834,932	1014,882
run 2	62,425	128,542	201,790	285,254	377,133	470,528	580,602	693,945	832,613	998,512
run 3	62,128	126,419	201,875	286,671	376,798	475,042	579,447	694,116	833,472	992,814
run 4	62,296	126,685	202,233	297,491	377,104	475,135	576,586	694,656	831,209	1003,178
run 5	62,264	126,478	202,617	292,049	376,949	475,893	580,141	717,653	832,724	1030,800
run 6	62,568	127,400	202,388	297,263	376,291	474,934	576,969	704,017	843,645	1095,868
run 7	62,014	126,804	202,100	295,357	375,959	473,488	577,969	800,602	844,967	1031,355
run 8	62,445	126,033	202,310	288,166	377,709	475,157	576,608	711,063	832,313	1000,328
run 9	62,640	126,138	202,163	285,051	376,713	466,289	579,976	694,170	835,698	996,791
run 10	62,414	127,287	202,322	285,000	377,206	472,443	578,037	696,483	831,676	1005,142
average	62,354	126,874	202,207	289,735	376,862	473,375	578,422	709,818	835,325	1016,967
SD	0,189	0,737	0,242	5,292	0,489	2,966	1,512	33,042	4,937	30,851

Table 19: Test 4: 8 parallel tasks.

Run \ # elem.	110	210	310	410	510	610	710	810	910	1010
run 1	22,715	38,746	59,006	87,275	116,766	147,661	184,650	225,821	264,597	310,191
run 2	21,685	39,066	58,437	86,014	115,590	150,020	182,513	218,467	263,810	321,264
run 3	21,991	38,608	60,602	87,927	114,035	148,706	182,862	222,954	292,506	321,958
run 4	20,707	38,416	59,860	88,251	116,838	148,232	183,704	219,740	299,070	323,615
run 5	21,334	36,873	60,499	88,211	116,394	149,416	184,489	224,429	266,401	321,817
run 6	20,411	38,461	58,090	87,156	114,653	152,121	185,463	224,580	271,729	321,676
run 7	22,201	37,327	60,208	87,105	116,801	153,807	181,580	222,303	282,353	322,143
run 8	19,252	38,576	58,674	85,921	114,838	149,332	185,319	221,703	268,215	324,424
run 9	21,147	37,355	59,468	88,035	117,341	146,240	186,913	222,318	267,311	325,642
run 10	20,359	39,240	59,342	84,696	113,070	145,748	185,634	219,980	264,962	324,549
average	21,180	38,267	59,419	87,059	115,633	149,128	184,313	222,230	274,095	321,728
SD	1,030	0,799	0,872	1,178	1,426	2,465	1,639	2,345	12,718	4,316

B.5 Results of 9 parallel generation instances

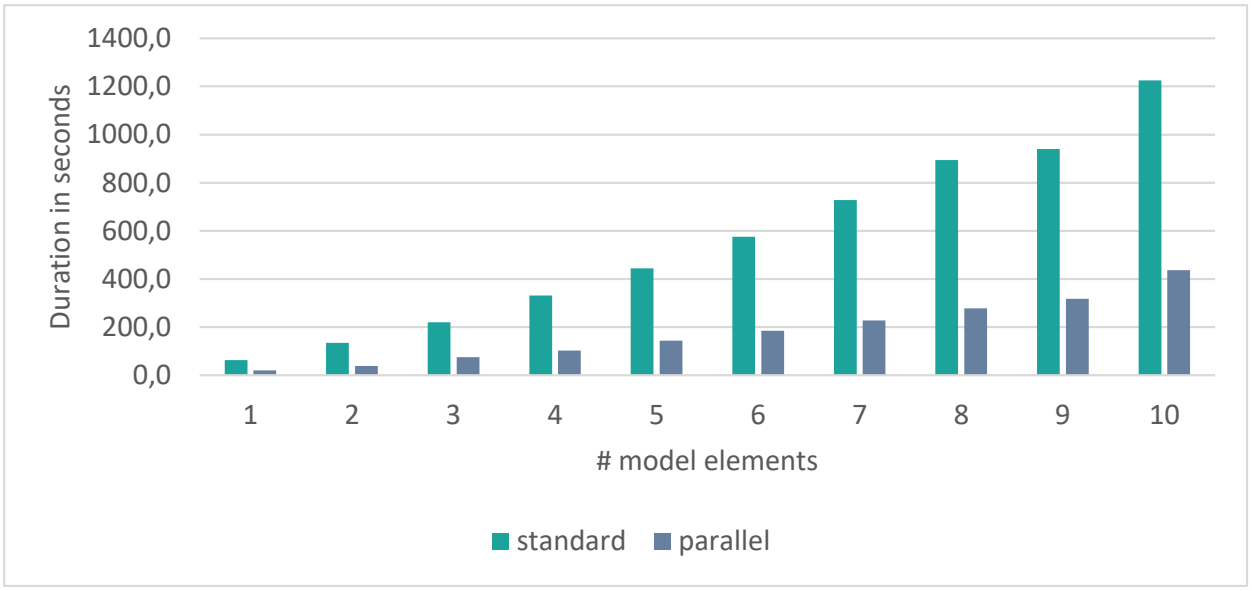

Figure 19: Generation duration chart with standard model generation compared to generation with 9 parallel generation instances.

Table 20: Test 5: baseline test.

# elem. / Run	110	210	310	410	510	610	710	810	910	1010
run 1	63,479	131,088	205,539	289,963	381,206	480,905	588,266	706,778	977,257	1005,953
run 2	63,824	129,650	206,297	289,591	383,696	479,498	590,711	702,504	839,904	1086,633
run 3	63,591	130,441	206,014	289,368	381,786	477,255	587,729	698,194	877,363	1026,001
run 4	63,039	130,511	206,274	289,177	382,782	478,904	585,849	698,638	962,817	1002,679
run 5	63,509	132,669	206,128	288,292	379,665	478,785	587,171	698,314	905,326	1003,724
run 6	63,057	130,416	206,373	290,012	381,475	478,052	586,221	697,625	936,121	998,815
run 7	63,347	129,833	205,733	288,723	381,754	478,303	588,163	699,317	995,881	998,345
run 8	63,366	131,049	206,207	290,171	380,636	477,895	585,653	695,540	870,342	999,442
run 9	63,640	129,898	205,936	289,076	380,160	477,984	587,894	694,419	841,115	1038,842
run 10	63,417	129,816	206,244	288,015	383,230	477,242	588,506	700,572	836,419	1168,711
average	63,427	130,537	206,075	289,239	381,639	478,482	587,616	699,190	904,255	1032,915
SD	0,264	0,940	0,294	0,661	1,236	1,125	1,659	3,484	56,113	30,223

Table 21: Test 5: 9 parallel tasks.

# elem. / Run	110	210	310	410	510	610	710	810	910	1010
run 1	20,021	36,764	63,442	89,722	119,770	158,699	196,201	237,656	280,604	324,350
run 2	23,671	40,377	61,938	85,778	122,824	149,326	191,572	234,930	275,859	318,257
run 3	21,025	39,450	61,609	87,901	118,738	151,839	191,082	232,500	277,355	324,262
run 4	22,744	35,111	64,256	87,384	118,490	156,825	192,616	230,265	270,737	325,768
run 5	21,462	40,434	61,948	88,613	118,784	151,790	186,635	229,285	275,698	322,757
run 6	23,102	36,965	60,976	90,758	119,635	155,109	191,344	234,099	271,106	315,929
run 7	21,250	40,383	64,454	88,519	118,295	152,053	191,606	234,536	274,480	323,991
run 8	22,914	41,535	63,892	91,516	117,614	155,105	192,196	235,752	269,692	316,575
run 9	21,503	37,631	62,312	89,751	118,532	158,733	194,457	232,603	271,757	313,306
run 10	19,989	34,655	62,301	88,878	119,080	152,209	190,591	234,601	296,230	314,910
average	21,768	38,331	62,713	88,882	119,176	154,169	191,830	233,623	276,352	320,011
SD	1,260	2,288	1,346	1,856	1,596	3,105	2,605	2,801	3,731	3,920

B.6 Results of 10 parallel generation instances

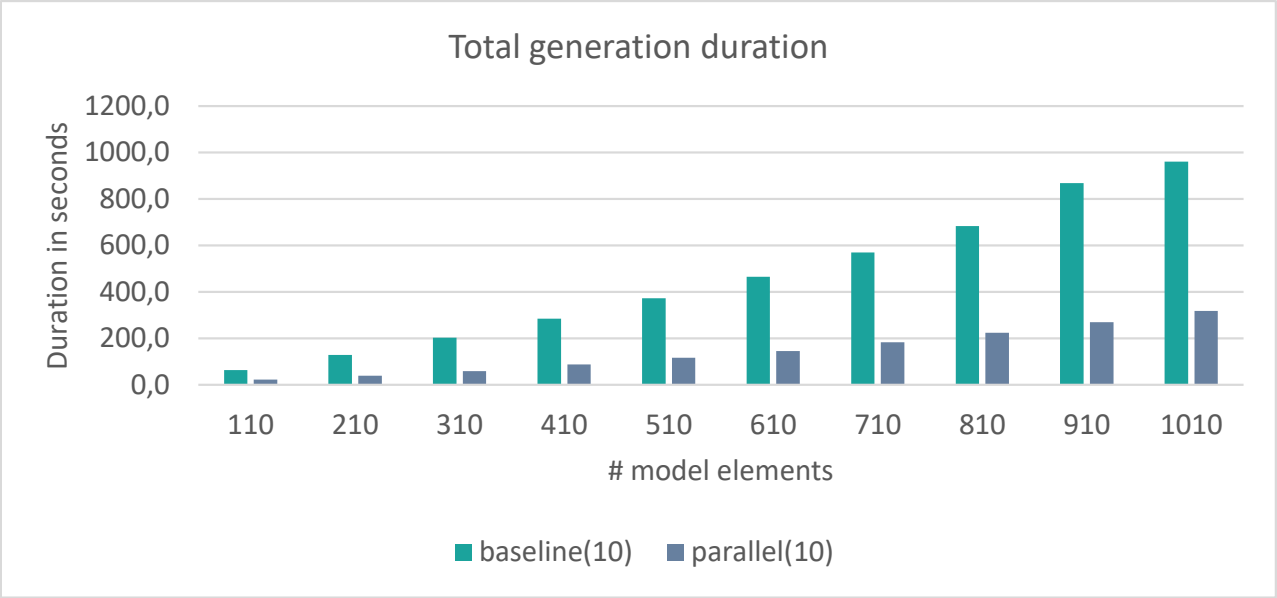

Figure 20: Generation duration chart with standard model generation compared to generation with 10 parallel generation instances.

Table 22: Test 6: baseline test.

Run \ # elem.	110	210	310	410	510	610	710	810	910	1010
run 1	61,79	126,715	204,036	285,613	371,922	468,335	690,52	699,078	840,018	990,717
run 2	62,283	127,917	203,006	283,893	370,504	467,949	596,884	699,63	834,587	988,115
run 3	61,758	126,861	203,917	285,353	370,695	469,567	653,44	698,747	837,629	987,045
run 4	62,194	127,434	202,909	283,155	371,024	470,780	661,29	702,366	837,515	985,1
run 5	61,951	127,052	202,957	282,273	372,312	470,791	604,024	696,394	837,507	989,654
run 6	62,530	126,87	203,412	282,814	371,610	469,327	655,376	698,78	837,519	990,194
run 7	62,398	126,892	202,293	282,466	370,684	469,352	610,325	694,883	836,253	990,001
run 8	62,509	132,126	204,029	283,249	372,264	469,156	626,155	699,179	833,507	990,477
run 9	62,481	127,519	202,452	281,281	371,017	470,785	619,431	695,069	835,918	981,508
run 10	62,362	127,964	203,794	281,124	371,478	469,991	665,879	698,968	835,268	984,895
average	62,226	127,735	203,281	283,122	371,351	469,603	638,332	698,309	836,572	987,771
SD	0,310	1,818	0,636	1,265	0,741	1,012	32,924	2,221	2,022	1,991

Table 23: Test 6: 10 parallel tasks.

Run \ # elem.	110	210	310	410	510	610	710	810	910	1010
run 1	24,219	44,614	70,707	97,591	129,228	160,618	203,504	263,939	304,225	351,458
run 2	24,039	39,769	67,364	92,971	129,429	164,996	212,706	253,374	323,773	350,301
run 3	26,424	42,923	65,721	99,678	130,120	162,997	205,732	266,384	308,812	349,740
run 4	24,417	43,464	67,848	94,150	125,754	165,411	201,646	253,964	304,248	355,930
run 5	24,122	44,076	65,118	98,179	128,910	166,755	201,749	249,259	306,750	345,912
run 6	25,851	40,983	68,906	93,757	125,770	163,342	206,661	251,962	306,985	354,070
run 7	21,570	43,826	66,757	98,229	123,331	164,835	206,157	255,584	315,216	347,628
run 8	24,225	41,095	69,284	98,406	131,225	160,903	203,184	253,753	303,024	359,164
run 9	26,098	44,335	67,910	100,520	125,094	162,643	254,657	250,062	306,718	346,915
run 10	24,716	43,137	67,726	98,027	123,252	163,921	224,933	255,959	299,984	339,143
average	24,568	42,822	67,734	97,151	127,211	163,642	212,093	255,424	307,974	350,026
SD	1,438	1,752	1,875	2,567	2,701	2,177	3,608	5,962	7,041	4,393

B.7 Results of 11 parallel generation instances

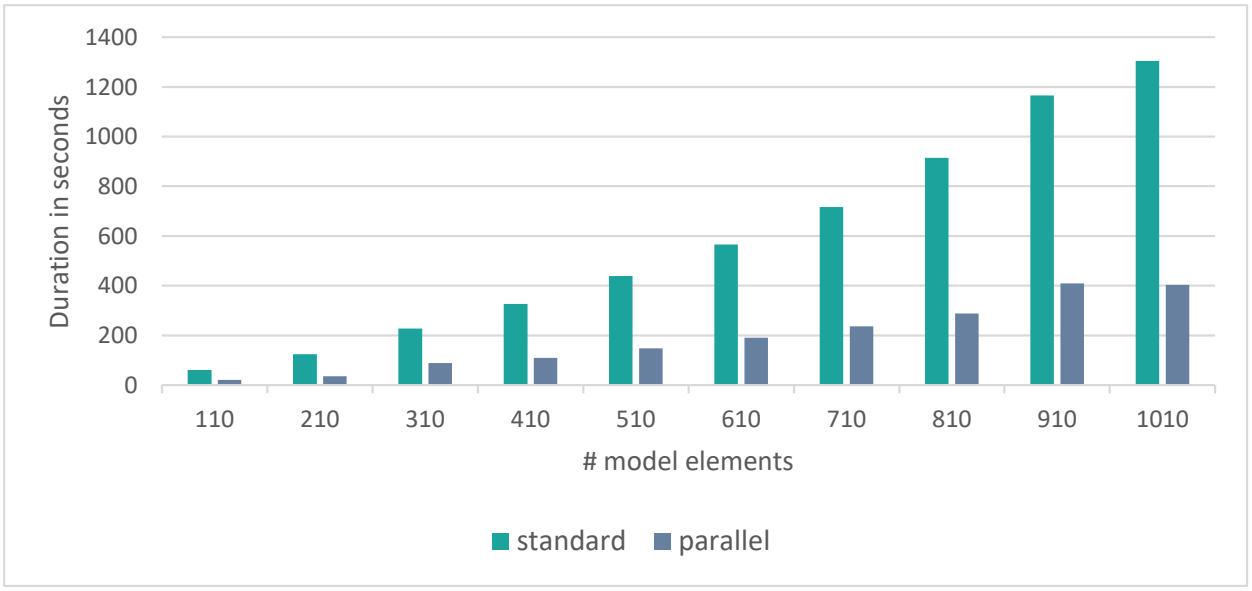

Figure 21: Generation duration chart with standard model generation compared to generation with 11 parallel generation instances.

Table 24: Test 7: baseline test.

# elem.	110	210	310	410	510	610	710	810	910	1010
run 1	72,873	149,378	235,597	338,688	453,112	576,501	735,861	901,663	811,104	962,083
run 2	72,811	148,429	235,862	335,697	446,206	572,724	725,869	999,396	813,272	1202,636
run 3	72,747	146,223	234,599	338,285	454,305	577,257	728,004	894,402	807,8	959,232
run 4	72,511	149,204	236,226	335,466	445,255	579,606	733,632	902,925	811,151	961,398
run 5	72,162	146,471	234,863	337,564	439,788	576,17	726,808	899,768	808,679	966,088
run 6	73,191	148,846	236,044	336,796	442,457	574,318	730,188	899,255	811,715	964,278
run 7	72,3	147,342	237,656	337,734	442,672	577,26	731,812	899,822	808,675	1122,096
run 8	72,986	148,824	238,366	335,044	441,39	580,876	731,447	928,913	815,761	1048,394
run 9	72,83	148,684	234,689	334,993	446,578	574,637	731,369	1033,643	815,037	1150,409
run 10	72,588	147,891	235,214	335,935	444,309	576,986	731,573	993,408	817,101	1111,74
average	72,700	148,129	235,912	336,620	445,607	576,634	730,656	935,320	812,030	1044,835
SD	0,349	1,242	1,290	1,374	5,380	2,623	3,434	35,395	2,657	93,345

Table 25: Test 7, 11 parallel tasks.

# elem.	110	210	310	410	510	610	710	810	910	1010
run 1	23,816	52,589	59,323	116,171	150,431	193,761	241,716	277,218	289,33	329,334
run 2	27,129	44,543	79,022	111,545	155,041	175,254	241,412	287,09	274,16	326,213
run 3	24,318	47,661	79,101	109,088	145,071	188,409	236,836	290,805	269,309	328,686
run 4	26,955	44,425	76,75	112,164	148,988	186,555	238,098	289,826	283,918	323,777
run 5	32,291	43,972	76,314	111,433	129,053	191,062	237,498	287,014	281,822	331,324
run 6	26,146	44,345	77,034	115,123	149,206	189,383	239,122	284,278	282,903	329,986
run 7	26,939	44,865	76,66	112,01	150,917	189,247	233,681	287,743	273,961	322,537
run 8	31,779	44,08	73,541	112,38	148,817	189,907	239,136	292,883	279,991	323,678
run 9	23,803	44,207	75,334	111,093	155,424	188,873	234,594	286,771	274,796	328,704
run 10	26,929	44,371	73,616	114,056	150,273	190,13	231,942	284,136	283,643	326,622
average	27,011	45,506	74,670	112,506	148,322	188,258	237,404	286,776	279,383	327,086
SD	3,105	2,984	6,458	2,217	7,832	5,534	2,584	4,790	6,507	3,335

B.8 Results of 5 parallel generation instances with a small model

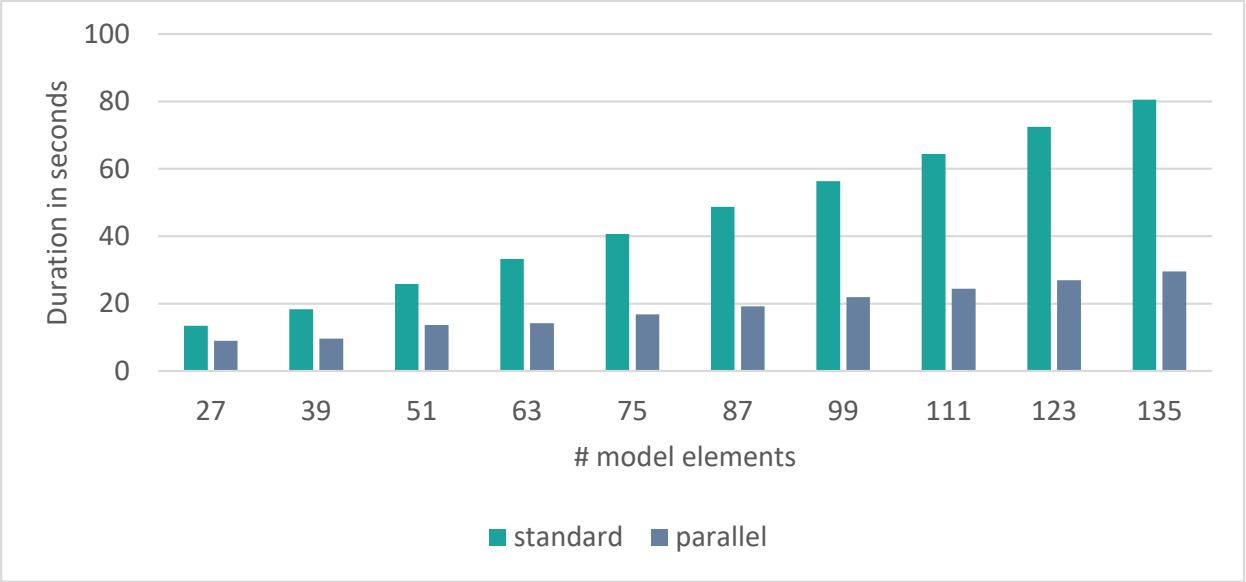

Figure 22: Generation duration chart with standard model generation compared to generation with 5 parallel generation instances for a smaller amount of model elements.

Table 26: Test 8, standard model generation.

# elem.	110	210	310	410	510	610	710	810	910	1010
run 1	13,299	19,178	25,694	33,24	40,674	49,016	56,336	64,808	72,178	80,859
run 2	13,424	18,274	25,932	33,23	40,734	48,82	56,352	64,12	72,244	80,439
run 3	13,422	18,274	25,813	33,238	40,738	48,604	56,149	64,436	72,732	80,424
run 4	13,459	18,22	25,69	33,132	40,722	48,744	56,095	64,399	72,688	80,383
run 5	13,453	18,208	25,932	33,311	40,627	48,757	56,121	64,698	72,478	81,018
run 6	13,452	18,36	25,739	33,299	40,669	48,604	56,092	64,455	72,614	80,691
run 7	13,41	18,198	25,757	33,265	40,803	48,746	56,585	64,395	72,444	80,72
run 8	13,423	18,281	25,92	33,236	40,707	48,497	56,446	64,335	72,552	80,091
run 9	13,435	18,31	26,089	33,278	40,534	48,856	56,421	64,546	72,612	80,013
run 10	13,507	18,25	25,816	33,02	40,89	48,777	56,339	64,205	72,326	80,196
average	13,428	18,355	25,838	33,225	40,710	48,742	56,294	64,440	72,487	80,483
SD	0,060	0,329	0,105	0,055	0,053	0,159	0,185	0,213	0,199	0,299

Table 27: Test 8, parallel model generation.

# elem.	110	210	310	410	510	610	710	810	910	1010
run 1	9,059	9,411	14,207	14,229	17,221	19,344	21,654	24,271	27,033	29,635
run 2	8,969	9,568	13,754	14,556	16,292	18,844	21,801	24,522	26,857	29,26
run 3	9,239	9,437	13,763	14,212	16,743	19,536	21,757	24,29	27,38	29,546
run 4	8,828	9,987	13,06	14,425	17,301	19,301	21,637	24,627	26,33	29,112
run 5	9,004	9,996	13,542	14,642	16,184	19,65	22,071	24,452	27,219	29,867
run 6	9,484	9,606	13,586	14,273	16,696	19,394	21,581	24,208	26,813	30,396
run 7	9,11	9,384	12,376	14,342	16,984	19,085	22,033	24,876	26,815	29,341
run 8	9,029	9,873	12,865	14,396	16,621	19,452	21,935	24,74	26,873	29,542
run 9	9,28	10,072	13,633	14,352	16,99	18,816	21,218	24,113	26,608	29,069
run 10	9,064	9,14	13,775	14,311	17,382	19,449	21,68	24,482	26,902	29,515
average	9,107	9,647	13,456	14,374	16,841	19,287	21,737	24,458	26,883	29,528
SD	0,232	0,258	0,587	0,154	0,402	0,257	0,186	0,239	0,314	0,402

B.9 Results of 4 parallel generation instances with a small model

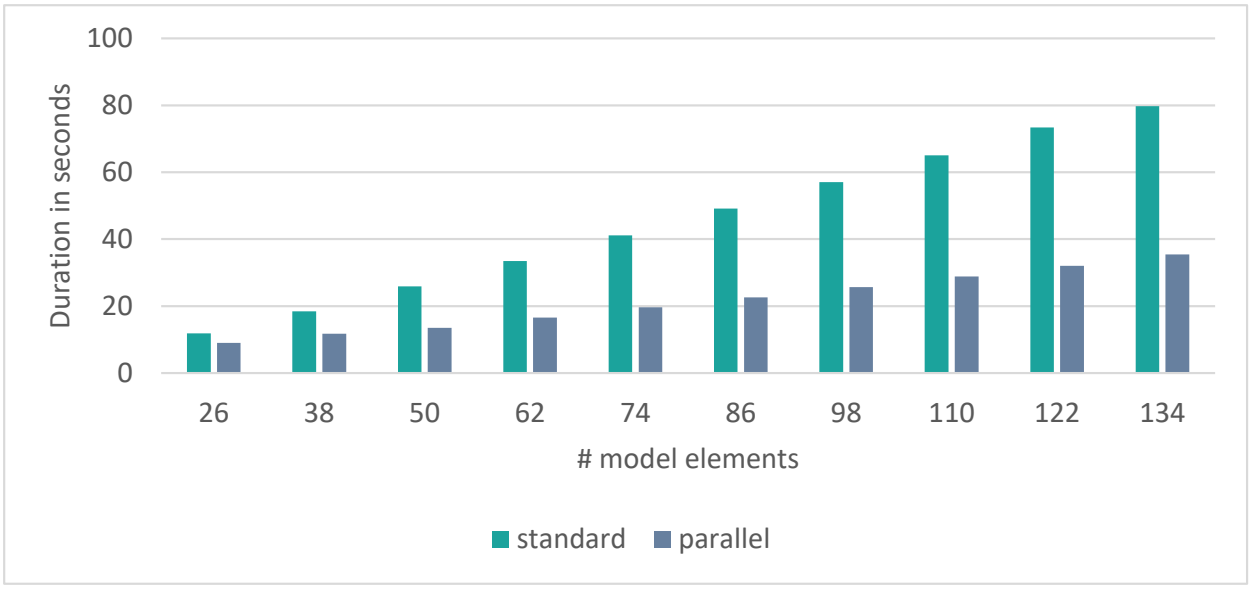

Figure 23: Generation duration chart with standard model generation compared to generation with 4 parallel generation instances for a smaller amount of model elements.

Table 28: Test 9, standard model generation.

# elem.	110	210	310	410	510	610	710	810	910	1010
run 1	11,727	18,476	25,768	33,408	41,448	49,085	57,159	65,571	73,702	81,338
run 2	11,842	18,351	25,946	33,501	40,979	49,196	56,937	65,146	73,044	79,318
run 3	11,833	18,5	25,894	33,281	41,153	49,264	57,094	64,98	73,731	82,041
run 4	11,904	18,432	25,789	33,372	41,337	49,451	56,844	64,786	73,262	79,455
run 5	11,79	18,503	25,875	33,429	41,153	49,051	57,085	65,071	73,795	79,971
run 6	11,794	18,401	26,081	33,511	41,169	48,79	57,392	64,887	73,258	78,814
run 7	11,899	18,424	25,889	33,444	41,106	49,13	57,08	65,606	73,377	79,422
run 8	11,909	18,387	25,878	33,517	41,404	49,005	57,201	65,014	73,35	79,326
run 9	11,869	18,382	25,952	33,488	41,265	49,265	56,834	65,078	73,515	78,974
run 10	11,977	18,519	26,076	33,446	40,93	49,116	57,049	64,901	73,198	79,171
average	11,854	18,438	25,915	33,440	41,194	49,135	57,068	65,104	73,423	79,783
SD	0,060	0,055	0,096	0,080	0,161	0,194	0,165	0,302	0,271	1,128

Table 29: Test 9, parallel model generation.

# elem.	110	210	310	410	510	610	710	810	910	1010
run 1	9,672	11,639	13,602	16,626	19,396	22,571	26,045	28,485	31,561	35,152
run 2	9,04	11,698	13,586	16,538	19,99	22,783	25,92	28,665	32,205	35,438
run 3	8,961	11,628	13,594	16,613	19,653	22,451	25,557	28,847	32,501	35,275
run 4	8,939	11,64	14,05	16,316	19,591	22,807	25,267	28,786	32,467	35,647
run 5	9,011	11,883	13,69	16,536	19,855	22,704	25,443	30,071	31,861	35,548
run 6	8,959	11,737	13,356	16,696	19,469	22,661	25,458	29,072	32,1	35,313
run 7	8,859	11,688	13,57	16,362	19,651	22,612	25,964	28,533	32,036	36,273
run 8	9,072	11,609	13,305	16,473	19,462	22,845	25,652	28,509	31,932	35,897
run 9	9,046	11,757	13,404	16,663	19,471	22,688	25,315	28,442	32,176	34,943
run 10	8,944	12,482	13,469	16,815	19,517	22,543	25,8	28,924	31,974	35,449
average	9,050	11,776	13,563	16,564	19,606	22,667	25,642	28,833	32,081	35,494
SD	0,284	0,089	0,226	0,131	0,204	0,133	0,283	0,524	0,313	0,369

B.10 Results of 3 parallel generation instances with a small model

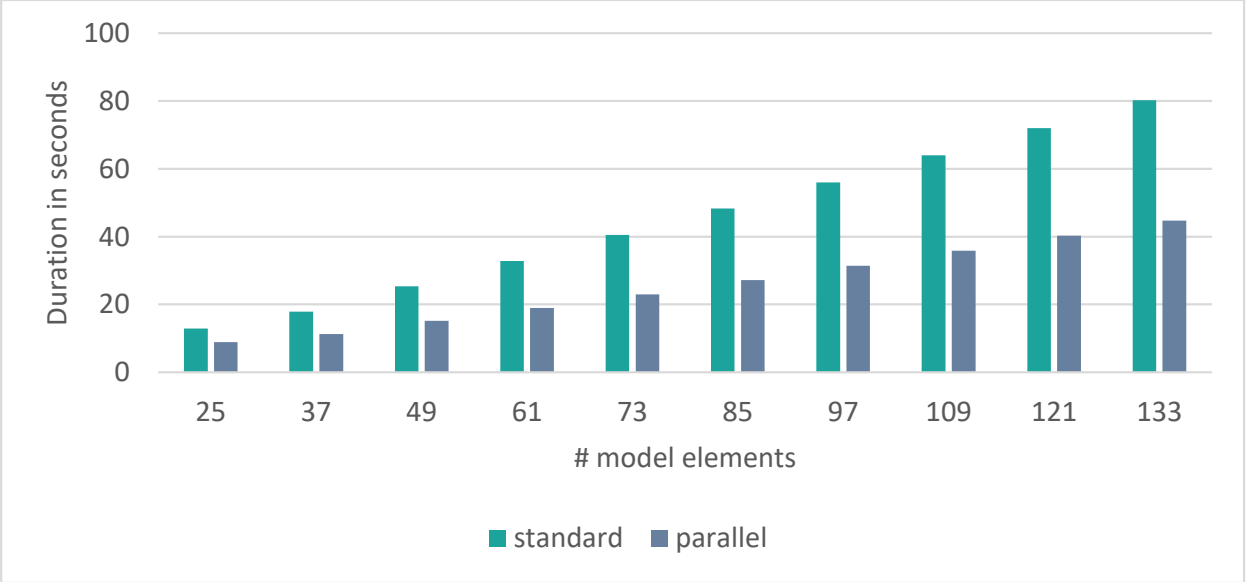

Figure 24: Generation duration chart with standard model generation compared to generation with 3 parallel generation instances for a smaller amount of model elements.

Table 30: Test 10, standard model generation.

# elem.	110	210	310	410	510	610	710	810	910	1010
run 1	12,756	17,742	25,355	33,022	40,575	48,301	56,376	63,89	71,949	79,85
run 2	12,911	17,841	25,35	32,806	40,73	48,157	56,265	64,198	72,307	80,108
run 3	12,901	17,793	25,284	32,907	40,757	48,32	56,045	63,929	72,101	80,128
run 4	12,838	17,895	25,347	32,762	40,252	48,196	55,979	64,076	71,882	80,31
run 5	12,857	18,179	25,403	32,744	40,27	48,071	55,743	63,79	72,067	81,016
run 6	12,833	17,97	25,368	32,887	40,552	48,129	56,185	64,248	72,015	80,091
run 7	12,867	17,871	25,27	32,893	40,545	48,08	55,956	64,024	71,83	80,181
run 8	12,849	17,85	25,563	32,89	40,288	48,409	55,866	64,04	71,915	80,211
run 9	12,909	17,824	25,462	32,854	40,463	48,51	55,807	64,242	71,903	79,917
run 10	12,812	17,884	25,464	32,769	40,603	48,403	56,113	63,91	71,924	80,111
average	12,853	17,885	25,387	32,853	40,504	48,258	56,034	64,035	71,989	80,192
SD	0,056	0,134	0,090	0,090	0,203	0,123	0,211	0,154	0,152	0,341

Table 31: Test 10, parallel model generation.

# elem.	110	210	310	410	510	610	710	810	910	1010
run 1	8,994	11,402	15,108	19,006	23,045	27,058	31,546	35,762	40,269	45,038
run 2	8,884	11,425	15,214	19,117	23,005	27,498	31,306	36,152	40,157	44,621
run 3	8,88	11,259	15,168	19,08	22,931	26,993	31,245	35,597	40,828	44,747
run 4	8,857	11,171	15,242	19,121	22,944	27,253	31,538	35,624	40,39	45,036
run 5	8,941	11,161	15,084	18,859	23,113	27,256	31,425	35,768	39,974	44,509
run 6	8,803	11,343	15,16	19,332	23,033	27,265	31,233	35,676	39,93	44,636
run 7	8,9	11,229	15,246	19,103	22,783	27,168	31,432	36,269	39,948	45,161
run 8	8,902	11,205	15,241	18,755	22,882	27,15	31,374	36,014	40,699	44,445
run 9	8,777	11,261	15,199	18,957	23,447	26,987	31,556	35,901	40,18	44,467
run 10	8,827	11,259	15,088	18,782	22,882	27,09	31,731	35,902	40,204	44,375
average	8,877	11,272	15,175	19,011	23,007	27,172	31,439	35,867	40,258	44,704
SD	0,067	0,103	0,063	0,177	0,104	0,154	0,121	0,254	0,344	0,270

B.11 Parallel generation durations combined

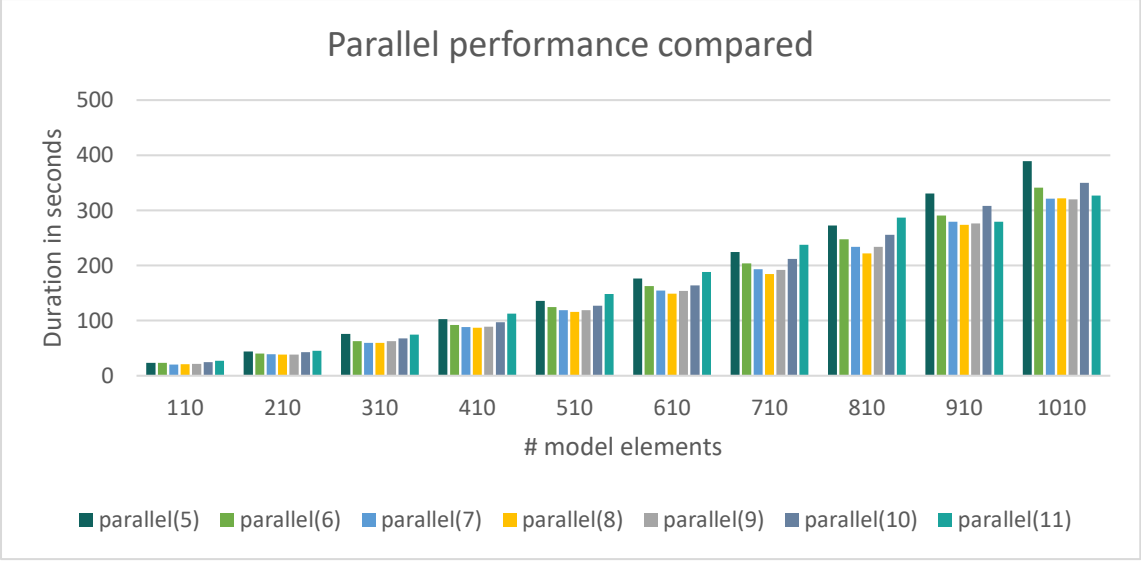

Figure 25: Parallel generation durations of 5 to 11 parallel instances combined.

B.12 Overhead of loading the source model with 11 parallel instances

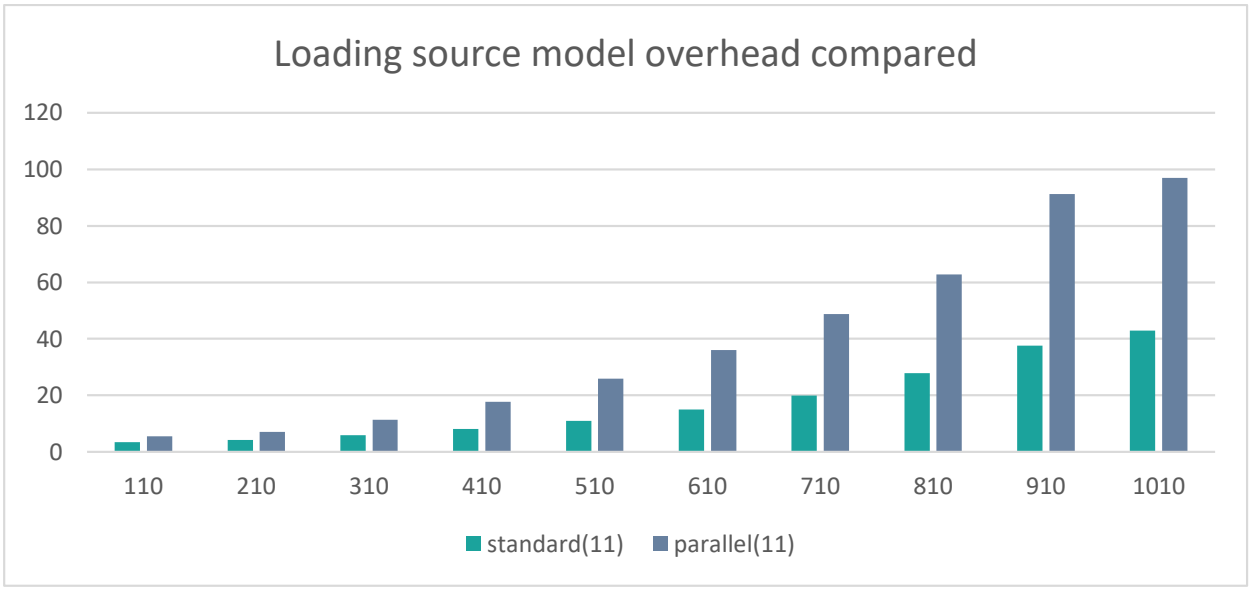

Figure 26: Source model loading overhead with 11 parallel instances compared to sequential model transformation.

C Paper

Towards a solution for improving the performance of model transformations in Model-Driven Development

Bart Smolders*†, Michiel Overeem† , Slinger Jansen*, and Jan Martijn E. M. van der Werf*
* Utrecht University, Princetonplein 5, 3584 CC Utrecht, Netherlands
Email: {j.m.e.m.vanderwerf, slinger.jansen}@uu.nl
†AFAS Software, Philipsstraat 9, 3833 LC Leusden, Netherlands
Email: {b.smolders, m.overeem}@afas.nl

***Abstract*— Utilizing models in software engineering is gaining popularity and there is an increasing need to generate faster. There are many approaches available to fulfill this need by leveraging the Atlas Transformation Language (ATL) model transformation language and its underlying platform to improve the performance of a generator. However, these approaches are specific for declarative transformation languages, like ATL. In this thesis we do a knowledge analysis of several such approaches because of the good availability. These are ATL on MapReduce, multi-threaded ATL, live transformations and lazy transformations. All of these approaches rely on model element traceability which is why traceability approaches are also analyzed. Our case study organization developed a generator where the model transformation are programmed in a general-purpose language. This limits the applicability of the available approaches in literature which is why a more generic approach is researched that improves the performance of model transformations. A reference architecture is proposed with corresponding documentation. This documentation describes important design decisions, required protocols and practical issues that must be taken into account or addressed when developing a generator aimed at faster model transformation. These design decisions and processes result from the performed case study where we implement approaches from the knowledge analysis. First, traceability is implement followed by partial model transformation that use the insights gathered from traceability. Both are prerequisites from parallel and incremental model transformation which are common approaches to reduce the time needed to finish a task. Finally, an experiment is performed with parallel model transformation which shows a performance improvement of 2.5 to 3.5 times.**

I. Introduction

Utilizing models in software engineering is gaining popularity and there is an increasing need to generate faster, since end-users want fast responding applications [8]. However, organization have difficulties achieving this. [9] mention that industrial validation of available literature on MDD is scarce and can thus only help organization marginally. Moreover, there is currently a lack of any scientific artifact that presents information to create an optimized generator in a structured way. This artifact could greatly help MDD adopting organizations in building high performing generators. Literature is available on declaratively model transformations such as Atlas Transformation Language (ATL). However, there is only very little literature available on generators built in a general-purpose language in combination with improving performance. The case study organization encountered this issue in their quest in reducing the generation time.

Figure 1 shows a typical MDD execution flow where an application is created given an input model. Executing this flow typically results in unnecessary waiting time for end-users since all transformations are executed sequentially. The total generation time may take minutes up to hours [14] which negatively influences the user experience as stated by [4]. For Software Producing Organizations (SPOs) it is therefore crucial to improve this process to reduce the time required to create the initial application and is the first identified problem.

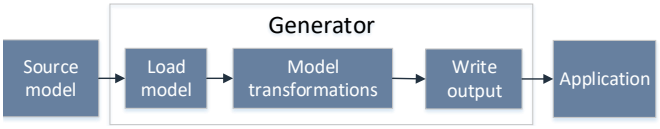

Fig. 1: Generator architecture of the case study organization. The arrows show the data flow between two stages. The transformation pipeline itself is not parallel but sequential.

Furthermore, SPOs may offer end-user variability as explained in the introduction where the end-user manages the model and can thus make modifications at any time. Additionally, the end-user determines the moment when a regeneration of the model is triggered. The end-user would normally choose to directly trigger a regeneration after a model modification to apply the changes. This situation is opposed to when the organization that build the generator is in control of the model and can choose a point in time to upgrade the applications. These subsequent model generations form a second common use case which is called incremental generation. [11] argues that incremental refinement is very useful since it is a process that is executed far more frequent than a full generation. An implicit issue with large organizations and MDD is the high capacity of hardware resources needed for generating the applications. In case only a small portion of the model is

This is an AMUSE paper. See amuse-project.org for more information.

modified there are still many resources needed when the full model is generated. This also means that a larger amount of hardware resources must be available to handle peak moments to ensure high responsiveness. Cloud-scaling cannot provide a solution because of the overhead of spawning additional servers. This issue can be solved by using incremental model transformation.

The subject for our research is the imperatively programmed generator provided by the case study organization. The goal is to improve the performance of model transformations. The partial generation technique is used as a means to scope the research. This paper answers the question: *How can the performance of imperative model transformations be improved?*. The following sub research questions (SRQ) are designed to answer the main research question. First, available literature on improving the performance of model transformation is gathered and analyzed to answer **SRQ1**: *What is the current state of methods that reduce model transformation time in Model-Driven Development?*. Research topics are parallel generation to reduce generation time for the initial generation and incremental generation for subsequent generations. Moreover, traceability and partial generation are researched which are considered prerequisites for both parallel and incremental generation. Next, insight into the model transformations is needed to further research and reason about applicable improvements and answers **SRQ2**: *How is insight gained in the model transformations in a Model-Driven Development environment?*. With the insights available the second prerequisite for both incremental and parallel generation is researched, namely partial generation. This research answers **SRQ3**: *How can these insights be utilized to improve the model transformation performance in a Model-Driven Development environment?*. Research from SRQ2 and SRQ3 can be combined to handle both the initial and the subsequent generation scenarios. This is accomplished by utilizing parallelism and incremental generation and answers **SRQ4**: Does the utilization of the insights lead to an improved performance?

The remainder of the paper is structured as follows:

II. Related Work

A. Parallel Model Transformation

[13] research the scalability of model manipulation tools by using parallel execution. They mention that it is complex to implement parallel model transformation in a general-purpose language even though parallelism is a traditional way of scaling computations. The reason for this is the lack formalizations. Furthermore, they argue that ATL and the like have strong parallelization properties which is supported in the literature mentioned before. Furthermore, the default ATL compiler and virtual machine are adapted to support multi-threaded processing. The authors include task parallelism and data parallelism into their research as two common approaches to scale model transformations, both are explained next.

1) Task Parallelism: Each task processes the same and complete data set, but only executes a distinct set of the operations on that data. This approach works well in the case that no dependencies are introduced between tasks as a result of processing the same data set. These dependencies are introduced when tasks rely on the output created by other tasks to complete their computations as stated by [6]. This concept is of course not specific for languages like ATL but also for imperative languages as is demonstrated in [12]. Transformation rules written in ATL are highly independent from other rules which makes this approach suitable for implementing parallelism.

2) Data Parallelism: Data parallelism is the opposite approach in which the model is partitioned and distributed to the tasks. All the tasks execute the same set of operations on the assigned model partition. The model partitioning thereby determines how the transformation is distributed for parallel execution as explained by [15]. A main goal of this approach is to reduce cross-task interaction by reducing or eliminating the access of shared model elements. This is especially an interesting approach in a distributed setting where large models are processed and the communication cost between processing unit become high. This approach correspond with the ATL with MapReduce approach in the previous paragraph.

B. Huchards Traceability Framework

Trace info can help in these scenarios since model transformation can be very complex. MDD generator like in ATL and the case study generator are composed of multiple successive model transformations and it is therefore difficult for developers to recognize the flow of a single element and where it ends up in the output. To gather such traces a system is needed that collects trace information on the behavior of model transformations while they are executed. This concept is called a traceability framework in literature and such a framework is proposed by [7]. In their research a metamodel for trace information is provided and explained how to collect trace information. They use Kermeta to write model transformations which has both object-oriented and model-oriented constructs and use imperative structures. Moreover, they note that it is difficult to create traces for these model transformations because of the imperative syntax of the Kermeta language. The authors propose the following definitions:

- **Object** *An Object is the most general element and forms the base of all other elements in the Kermeta language.*
- **Link** *A Link references one source and one target* ***Object****.*
- **Step** *A Step contains the traces of a single transformation and are composed of multiple* ***Link****.*
- **Trace** *A transformation trace is a bipartite graph. The nodes are partitioned into two categories: source nodes and target nodes.* ***Step*** *can be chained as an ordered set to from a transformation chain trace.*

III. Gain insight into model transformations

As already explained in Section II traceability is a prerequisite for both incremental and parallel generation. Dependency information is needed to partial source models and to determine what target elements need to be updated given a source model change. We first describe the case study method and

traceability technique used for the case study. Then the case study itself is explained.

A. Huchards Traceability Framework Implementation

Huchards traceability framework is implemented in the case study generator to validate the approach. This approach is considered best applicable because of the language independence. For the trace creation itself code must be added to the model transformation code and this process is called explicit tracing. Since model transformations are intertwined throughout the code it can be difficult to recognize a single atomic model transformation. Therefore, it is difficult to choose the locations in the code where trace generating code should be placed. Furthermore, while adding trace generating code it soon became arbitrary where this code should be added. In literature we did not came across a structured method on how to implement trace generating code into the model transformation code. The executed approach requires significant work to trace all relevant model transformation which becomes increasingly complex for larger generators with over 200.000 lines of code. This observation is summarized in **Observation 1**.

> **Observation 1**: Adding trace generating code to a MDD generator can be complex and time consuming. Transformations written in an imperative languages can be complex that also make it difficult to correctly add trace code. Declarative transformations are mostly one-to-one transformations which are straightforward to trace and can usually be handled by the underlying platform.

After having implemented trace generating code between the model transformation code it is important to keep the trace code up-to-date (**Observation 2**). Model transformation can be adapted over time and the corresponding trace generating code must be kept up-to-date. Such an adaption can be a straightforward deletion or can be an operation where some logic is changed regarding the model transformation where the trace object themselves keep their meaning. This process becomes more complex when the functional meaning of a traced object is altered. The developer would in this case have to determine whether the trace is still needed and/or whether additional traces have to be created in case a trace chain is interrupted.

> **Observation 2**: It is important to keep the trace code up-to-date to ensure that a trace model can be created and that captures all traces between model transformations. The trace model is considered incorrect when it does not represent all traces that should exist given the source model and model transformation and can in that case not reliably be used for partial generation.

For efficiency purposes we only want to gather useful trace information (**Observation 3**). We noticed that it is hard as a developer to determine which transformations have to be traced. There are situations where transformations do not provide additional information regarding traceability. This might be the case for a chain of one-to-one model transformation where no new complex dependencies are introduced and the existing dependencies can be induced.

> **Observation 3**: Eliminate redundant traces in the trace model. For the sake of fast generation times it is useful to not trace all model transformation when this is not necessary.

However, it can be difficult to recognize this type of transformation since extensive knowledge is needed about the entire flow of the generator. This becomes increasingly complex as the size of the MDD generator grows. Nonetheless, developers can be aided in this process by a visualization of the trace model which makes it easier to recognize redundant traces. Moreover, by analyzing the trace model a program can hint redundant trace links. A precondition of using these supporting tools is that a complete trace model must be created. This is needed since automatic analysis is only reliable when full knowledge of the domain is available, otherwise invalid conclusion could be made. This complete trace model can then based on analysis be slimmed down to reduce the amount of collected traces. However, this would lead to an impasse when model transformation in the generator are added, edited or deleted. These operation would likely alter the trace model and again the complete trace model is needed to determine whether the trace model is affected and what trace an safely be removed from the model.

Lastly, we store the trace model in a file to make it reusable during model evolution. Therefore, we researched what format can efficiently store the trace model since no literature could be found on which format can best be used to store the trace model in and resulted in **Observation 4**. XMI is de facto standard in declarative model transformation such as in ATL where all models are expressed in XMI. In the case study we serialized the in-memory trace model to JSON since the available XML serializer could not serialize our complex data structure. We argue that JSON is a good format to use since it is more compact than XML and might be faster to serialize and deserialize compared to XML. These properties should make it more suitable format to make sure that trace model export and import do not slow the pipeline more than it should.

> **Observation 4**: No literature is available on what format is best used to store the trace model in. Several common format, like XMI and JSON, can be used and each provide benefits as well as limitations. XMI is more expressive where JSON would normally yield a smaller file size. We consider this an important matter since reading and writing an additional model next to the source models themselves imply also additional computation time.

IV. Use the gathered insights to improve model transformation performance

The insights can be used to partition the source model and is described in Section IV-A. Secondly, a partial generation technique from literature is implemented and researched on whether the approach is generalizable to other contexts in Section IV-B. Based on the insights gathered on the case generator (Section III) and the related work analysis the **data parallelism approach** from [13] is selected for implementing in the case study.

A. Model Partitioning Implementation

Model partitioning, which is needed for data parallelism, is not straightforward. The reason for this is that static analysis of the transformation is very complex for imperatively written transformations. [3] supports this claim by outlining that model partitioning is very challenging because of the many dependencies between model elements combined with complex transformation rules. Literature is provided on model partitioning for ATL transformations. However, there is insufficient applicable literature on the topic of model partitioning for non-declarative transformations. Since the case study generator does not possess formalizations like in declarative transformation only model partitioning of the source model can be researched.

1) Partitioning the case organization model: This part of the case study prepares the case organization source model for partial or parallel generation. Only the source model is analyzed since the transformation themselves cannot directly be analyzed in a formal way. A specific element type in the model is chosen to partition the model on. This element type can be seen as a module element to which other elements are attached. From the resulting partitioning we can conclude that the case input model was not designed with partial generation in mind. There are dependencies between work areas, moreover, there are also many internal dependency relations within a single work area. This makes it difficult to partition the model elements into equal partitions and this problem is recorded in **Observation 5**. This is generally desired because equal partitions result in the highest time reduction when doing parallel generation. That is, all partial instances should have more or less the same running time without any outlier which delays the generation and deployment flow. It must then, however, also be the case that the model element type distribution is approximately equal for the partitions since the generation time per element type can differ significantly.

> **Observation 5**: The structure of the source model significantly affects the resulting partitioning. A source model can usually be seen as a graph which means there are dependencies. The number of dependencies is ideally kept to a minimum to partition the model into equal parts. This is turn also reduce the number of overlapping elements in partitions.

Furthermore, the following questions were encountered during this case study for which we nor literature could provide any answer to. The questions are captured as separate observations and listed below.

> **Observation 6**: No guideline or protocol could be found to determine what dependencies are relevant for a given change. Related to that; how can this be leveraged to decrease the number of elements that need to be processed?
>
> **Observation 7**: No suitable element type or unit is found that is best used to do model partitioning. Furthermore; what properties does such an element type need to be effective and efficient for partitioning.

B. Partial Generation Findings

An issue is encountered that is a consequence of partial generation with a generator designed for sequential generation. Our sequential generator assumes the availability of all objects but is not the case for partial generation. An example of this is the phase where in-memory objects are compiled to a software component and is captured in **Observation 8**. To be able to compile a subset of the generated artifacts much more dependency analysis has to be performed. This means besides taking design-time and generate-time dependencies into account to also include run-time dependencies. However, this would make the partitioning phase even more complex. Moreover, the size of partitions will grow by significantly since much more elements are needed in the partition. This in turn reduce the number of parallel instances that can be used and increase the time needed to analyze the trace model for partitioning.

> **Observation 8**: When the MDD generator generates source code for a compiled language it cannot be compiled directly by that task itself. There are usually compile-time dependencies to artifacts that are not created by the current tasks and thus do no exist yet.

Moreover, a fundamental issue was encountered on how partial generation can be implemented and how to do it efficiently. The model partitioning algorithm require information that is needed to partition on. Other settings might be a set of model element to process or a toggle that indicates whether the entire model has to be generated. This information result from either pre-configured settings or from intermediate generator artifacts. This information could either be globally available or passed through the transformation pipeline up to the point where it is needed but no best practice is found and resulted in **Observation 9**.

> **Observation 9**: Make partitioning properties or information available to the partitioning algorithm. No efficient strategy is found for any architecture.

The partitioning algorithm is implemented directly after the source model loading model. This is required to ensure that all model elements are available during generation since there might be dependencies to elements outside the created partition. Moreover, the available generator architecture resulted in duplicate partitioning logic and is summarized in **Observation 10**.

> **Observation 10**: An early phase or a sufficiently high abstraction must be chosen to do the partitioning. Otherwise, partitioning code is spread throughout the model transformation pipeline.

V. Experiment with parallel generation

In this section the previously researched traceability and partial generation are combined to research both parallelism and incremental generation. By doing this we can provide an answer to **SRQ 4**: *Does the utilization of the insights lead to an improved performance?*. We start with parallel generation where we perform an experiment in Section V-C. After that,

we research the implementation of incremental generation in the case study generator.

A. Parallel Generation Rationale

Issues might arise regarding limited memory and high execution times when dealing with large and complex models as mentioned by [2]. [13] state that parallelism is a traditional way of scaling computations and is already extended to the context of MDD. Moreover, [1] show that the generation time can be greatly reduced by using multiple tasks over which the work is divided.

B. Data Parallelism Continued

Research on parallel generation continues with the data parallelism approach used earlier for research on partial generation. As was already stated in the related work section no modification to the generator itself is needed to use the approach for parallel model transformation. However, to be able to use our case generator in a parallel setup we had to duplicate the source model for each parallel task to avoid file locking conflicts. We define a parallel task as a process that use the generator executable with a set of arguments that runs in parallel with other instances of the generator. These argument are, among others, the location to the source model, the partition to generate and the output location where created artifacts are written to.

All partial generation tasks that are executed in parallel are generating their own output artifacts. However, in the case of duplicate filenames, for example, the actual contents of the files has to be merged. When this is the case each partial task needs a dedicated output folder to avoid file locking conflicts while writing to disk. Merging here comprise two operations, first the separate output folders must be copied to a single folder which is used by the application. Secondly, in case partial files are produced a merge operation must be executed. This merge often requires domain information to know how 2 or more files are to be merged into one. A merge as in version control systems does not work in this case since two partial files are not actual versions of each other.

> **Observation 11**: Partial output artifacts created by separate partial tasks must be merged to form a complete application. The merge operation of these files require domain knowledge such as what the content of the artifacts represent and how it should be merged.

C. Parallel Generation Experiment

The goal of the experiment is to prove a reduction of the total generation time and the overhead that parallelism might introduce. Additionally, this section describes how the experiment is conducted and how the results are measured and analyzed.

1) Initialization: For the experiments we created custom input models which are based on the models designed by the case organization. A single model is created for every test case and is composed of a predefined number of model elements.

All measurements are performed on a virtual machine (VM). The VM is installed on a native hypervisor together with two other VMs. The hypervisor runs on a host server with the following specifications:

- Microsoft Windows Server 2012 R2 Standard
- Processor: Intel Xeon CPU E5-2620 @ 2.40GHz v3, 8 Cores, 8 Logical Processors. L1 cache: 8 x 32KB. L2 cache: 8 x 256KB. L3 cache 8 x 15MB.
- Memory: HP DDR4-2133MHz 16GB.
- SSD: HP LK0800GEYMU 800GB.

The VM we used has the following specifications: 8 CPU cores, 16 GB RAM and 70GB SSD. Where possible all programs not needed for running the operating system or conducting the test are shut down. This is done to eliminate unnecessary load on the system. After every test there is a cool-down period to make sure any CPU activity of the previous test does not affect the next test.

2) Experiment Structure: The experiment is executed by conducting ten sub-experiments and each experiment consist of a baseline and hypothesis test which are described below. Each test is composed of ten test cases where every subsequent test case uses a model that contains a larger number of model elements. Each test case is repeated ten times (called a run) and the averages will be used for analysis. There is a cool-down period after a run of three minutes. In this interval period the hardware utilization will be restored to normal levels to make sure subsequent runs do not affect each other. A timings file is created while generating and contains the timings of each step in the model transformation pipeline of the generator and is used for later analysis.

a) Baseline Test: A baseline test is executed for each test case by generating the complete model with the unmodified generator. This version of the generator does not contains any changes made for this research like partial generation or traceability information collection. This generator version is suitable for baseline measurements since it is developed before our research started. Moreover, its properties were used to formulate the business problem of the case study company.

b) Hypothesis Test: These tests use the partial generation implementation from Section IV to run in parallel. Each partial tasks is passed a separate source model since a single model cannot be used in more than one program at the same time.

3) Analysis: Each sub-experiment results in two sets of ten total generation timings, one for the baseline and one for the hypothesis. The total generation time for the hypothesis test is the task with the longest running time. For the analysis itself we use the averages of the total generation timings. The graph with ten parallel tasks is shown in Figure 2 and depicts a time reduction given a certain number of model elements by using parallelism. All others tests show the same trend in performance improvement with a different number of parallel tasks. On average a generation time reduction of 2.5 to 3.5 times is achieved. This is slightly better than the results achieved with ATL on MapReduce as explained earlier. However, it must be noted that our models are not real world models and in practice such an equal distribution would be hard to accomplish due to model and transformation dependencies.

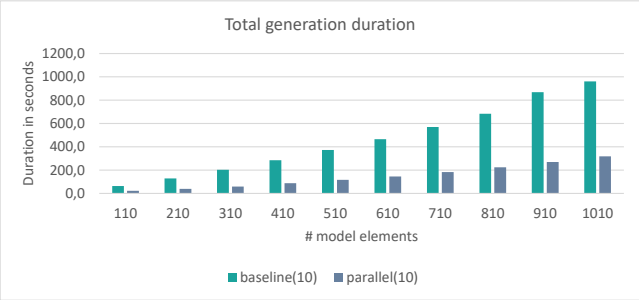

Fig. 2: Generation duration chart with standard model generation compared to generation with 10 parallel generation instances.

Moreover, overhead is introduced by, among others, running multiple generator tasks at the same time, as described above. However, the overhead is not directly visible from the results since we manually created the models and no partitions are computed. Nonetheless, the experiment results do show that the parsing operation becomes slower when generating in parallel compared to the baseline and is summarized in **Observation 12**. The disk is normally a common bottleneck but has only 10% utilization during the parsing phase. The probable bottleneck in this case would be the CPU which is nearing 100% utilization when multiple generator tasks are running at the same time. This observed trend holds for all transformations before the introduction of our partial generation construct. These transformations take up to 2 to 3 times longer in the hypothesis tests compared to their sequential counterpart. The experiment result that supports our observation is depicted in Figure 3.

Observation 12: Loading multiple source models at the same time adds transformation time to the model loading phase.

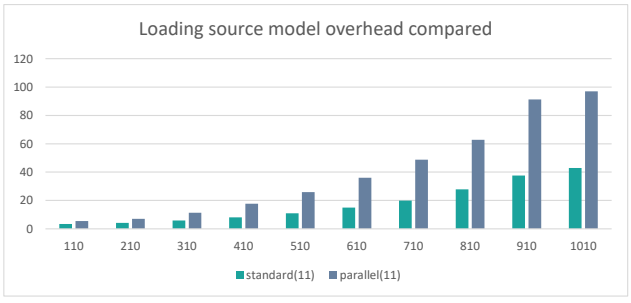

Fig. 3: Source model loading overhead with 11 parallel task compared to sequential model transformation.

Lastly, the separate steps in the pipeline can be compared where timings from the parallel tasks are aggregated. The standard generation is again used as the baseline and durations from the partial generations are compared against that. The aggregated timings compared to the baseline timings shows the overhead of parallelization as a whole.

D. Incremental Generation

Research on incremental generation is limited to literature study and to applying findings from the case studies to reason about possible issues incremental generation. No existing approach could be implemented due to the lack of a traceability model and the infrastructure needed to keep the application in memory is not readily available. The traceability case study provide us only with a small set of traces for only a few model transformations. It was too time consuming to add trace generating code throughout the complete generator as is elaborated in **Observation 1**. Despite not being able to implement an incremental model transformation approach we could reason about problems regarding the implementation and use an incremental transformation approach. The first issue we foresee in doing incremental generation is the process of calculating the delta of two model versions. In **Observation 12** we observed that is not desirable to load and parse multiple models at the same time. At some point a delta of two models must be calculated to use in incremental transformation. These two models are the updated model and the previous model version on which the changes are applied. We consider that is it useful to optimize this process since it would take unnecessary time and hardware resources to load and parse two complete sets of source model each time (see **Observation 13**).

Observation 13: Loading new and previous model to calculate a delta is time consuming. This delta-calculation approach is best avoided when possible to prevent introducing transformation overhead.

A second problem we expect regards the updating process of the trace model after the model is adapted which in turn invalidates the existing trace model (**Observation 14**). Model dependencies can be analyzed from the model itself but additional dependencies are introduced in the generator itself. The trace model can be updated by transforming the new model which automatically creates a new trace model as a result. However, this would defeat the purpose of using a trace model and incremental transformation since the transformation already created the complete application. Moreover, there is no applicable literature available that directly addresses this issue in our domain.

Observation 14: Efficiently creating a traceability model from an updated model is not straightforward. A native approach would execute the entire model transformation pipeline with the new model to collect all relevant traces for that model. However, this approach does not use any benefits of the incremental model transformation approach.

VI. Reference Architecture

This section describes our reference architecture that can be used to design a MDD generator tailored to fast model transformation. The references architecture is based on observations made in the case study described in the previous section. By proposing our reference architecture the main research question is answered: *How can the performance of imperative model transformations be improved?*

In this section the observations made previously are complemented with suggestions on how to deal the observed problem. Not all observations could be complemented with one or more suggestions when no solutions could be researched or due insufficient experience. providing suggestions for these observations is considered future research since we did not have the time to extensively research all observations. Moreover, we aim to provided suggestions that are also applicable to other generator architectures than the one we researched. This is needed to design the reference architecture is a more generic way and is not focused on the case study organization. Those suggestions are where possible based on literature, experiment results or domain experts from the case study company.

The guideline artifact itself is created by looking at the as-is situation in detail combined with conclusion and results from the case studies. An outline of the as-is situation of the case study company and is described in the research approach section. During the case studies we looked at shortcomings and bottlenecks of the current setup. Moreover, we primarily focused on architectural patterns and excluded detailed implementation specifics that might influence the model transformation performance. We could only make observations on the aspects we research and thus not all steps needed to create a MDD generator are explained in detail. Expanding our proposed reference architecture is left to future research where additional case studies provide new insights. In the remainder of this section we gradually built and explain our proposed architecture. All observations and suggestions made are numbered and linked to elements or arrows in the provided figures. First, the initial model transformation scenario is covered and depicted in Figure 4. After that, the incremental generation scenario is covered and shown in Figure 5.

A. Initial Model Transformation

1) Model Partitioning: The first artifact in Model-Driven Development flow are one or more source models (A). These source models contain dependencies and form the traceability model as explained in traceability case study (Section III). **Observation 5** states that the number of dependencies should be kept to a minimum since dependencies are disadvantageous when creating partitions with an equal distribution of model element types. We propose a solution in which the modeler is made aware of the dependencies he or she created either explicitly or implicitly by modifying the model. A source element relies on a specific property in the target element and thus any change that might modify this property also affects the source element. The modeler could be assisted while creating or modifying the source model about introducing new dependencies and the number of existing dependencies between two elements. It is beneficial to reduce the number of dependencies on properties from other elements. This can be achieved by following the dependency inversion idea of the SOLID programming principles by Robert C. Martin. This means for example that a base element is responsible for the calculation and a dependent element only requests the resulting value. This setup makes sure that a minimal amount of existing artifacts are recreated after a model modification.

Solution(s) for Observation 5: Make the modeler aware of the existing dependencies in the source model. Furthermore, indicate the impact a given model adaption has on the dependency or trace model. SOLID principles can be used to guide the modeler in designing the model that is best structured for model partitioning.

In general, loading the source model(s) into memory is the first operation performed by a MDD generator. This is needed before analysis and thus partitioning of the source models can be performed. We researched this phase and during our parallel generation experiment we observed that additional model transformation time in introduced in the model loading phase. This issue is elaborated in **Observation 12** and corresponds to B in Figure 4. We proposed two solutions to deal with the problem. First, start the pipeline sequentially where is single model is loaded and preparation steps are performed (B). This approach is still equivalent to the data parallelism approach. After that, the partition can be created (C) and executed in parallel in remainder of the pipeline (D). Secondly, in case there is an existing generation an external tool can be used that partitions the source models. Following this approach the existing generator architecture can be left unaltered and multiple instances of the generator can be executed with each a different model partition. It is the responsibility of the partition algorithm to create partitions that can be generated by itself without additional dependencies to other elements.

Solution(s) for Observation 12:

1) Postpone the model partitioning phase to ensure that only one instance of a source model has to be loaded which is used throughout the remainder of the generator. After the model is loaded it can be analyzed for partitioning and later parallel model transformation.
2) In case an existing generator is used an external tool can be leveraged that partitions the source models.

For both approaches see **Observation 1** and the corresponding solution on issues observed with collecting trace information used for model partitioning.

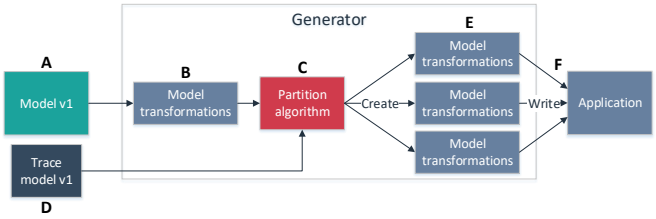

Fig. 4: Initial model transformation reference architecture. A source model (A) is passed to the generator. A first set of model transformation, among others, load the model into memory (B). Then a partition algorithm is executed that analyze the in-memory source model (C) and a trace model (D) to partition the model for parallel model transformation (E). The trace model is either based on involved model transformation given a source model or a model of all dependencies in the generator. Finally, each parallel model transformation task write a distinct set of output artifacts that together form the output application.

2) Create and Use Traceability: In **Observation 10** we observed that partition code was duplicated and used at multiple locations in the transformation pipeline. Before we already suggested a partition algorithm and therefore we also include this as a module in our reference architecture (C) to solve the observation. This module contains all partitioning logic in a central location and use the generated traceability model to partition the source models into a number of partitions. These partitions should be complete to make sure no external dependencies are present. These partitions are then generated in parallel (E).

> **Solution(s) for Observation 10**: We propose a model partitioning module which should be a single phase in the generation pipeline that partitions the source model(s). By conforming to this property we can also provide a solution to **Observation 12**, where we observed that the entire model is needed for each parallel instance. The created partitions are then given to the parallel generation instances to parallel model transformation.

We argue that the most important feature of an efficient partitioning algorithm is that small and evenly distributed partitions are created. This can be achieved by selecting the right model element type with the right granularity. This is easier said than done since no suitable element type or unit is found that is best used to do model partitioning. Furthermore, no element properties could be defined taht are needed to be effective and efficient for partitioning and is elaborated in **Observation 7**.

Creating the trace model (D) in a generator programmed in a general-purpose language is not straightforward. Moreover, this process is time-consuming since trace generating code is insert by every model transformation as mentioned in **Observation 1**. However, we could not find a solution that is less time-consuming and yields the same result. The traceability approach where trace generating code is added to model transformation requires that the transformations have to be executed with a source model in order to create a trace model. We consider this as a drawback since a large part of the transformation pipeline if not all is executed to collect all traces as described in **Observation 14**. We propose several solutions to create the trace model more efficiently. First, we describe a simulation approach where only a part of the transformation pipeline is executed up to a certain point. At this point all dependencies that exist are either known or can be deduced. This approach might require changes in the architecture to ensure that at some point only one-to-one transformation are executed of which the dependencies can be deduced. Since no new dependencies are introduced beyond this point there is no need to generate any further. It is of course desirable that this point comes as early in the transformation pipeline as possible. A re-run of the pipeline is needed with the new source model and the newly created trace model to generate the output artifacts themselves. Secondly, we propose an alternative method where model transformations are formalized to enable static analysis and thereby collecting trace information. A downside of this approach is that the abstract and imperative transformation rules must be kept in sync. A second issue is that all transformations must be extracted and rewritten in a formal language. A benefit of this approach is that the trace information is independent from source models and can be reused for all end user. Lastly, theory from abstract interpretation could be used to statically analyze the model transformations. [10] propose a new algorithm to analyze logic programs by using abstract interpretation. The algorithm focus on inferring dependencies between program expressions. However, no direct applicable approach is found for transformations programmed in a general-purpose language.

> **Solution(s) for Observation 14**:
>
> 1) A simulation approach where a part of the transformation pipeline is executed with the new model to gather traceability information. The pipeline is re-run to utilize the trace model for model partitioning to generate in parallel.
> 2) Formalize all model transformation and store them as a separate artifact. The formalization ensures that static analysis can be applied to create the trace info. The created trace model can be reused for all end users that use that version of the generator.
> 3) Use abstract interpretation to statically collect dependencies in model transformations.

3) Practical Take-aways: During our case study we researched how to best implement our partial generation algorithm. No literature was found and resulted in **Observation 9**. However, we could not provide any well-reasoned suggestions for the observation and is left to future research. Moreover, compiling the output artifacts separately did not work in D as observed in the partial generation case study and is elaborated in **Observation 8**. We describe a work-around on how to deal with this observed issue. The work-around is to avoid

generating source code for a compiled language. This is the approach our case study organization is already migrating to. This approach entails that the generator generates artifacts that are interpreted at runtime.

Solution(s) for Observation 8: Avoid generating source code for a compiled language in partial generation. This eliminates many dependencies between output artifacts. Instead generate artifacts that can be interpreted. Since the actual use of the artifacts are delayed to runtime all partial outputs are combined which means that all dependencies can be resolved.

Lastly, the partial model transformation tasks at D (Figure 4) all write their output to a single output location (F). In **Observation 11** we described the file locking conflict problem with this approach. Therefore, we propose a solution by ensuring, where possible, that the partial model transformation tasks to not produce overlapping output artifacts. This eliminates the need to copy different output folders and to merge files with the duplicate filenames. The internals of the generator must be adapted in such a way that each partial task creates a disjoint set of artifacts. This can primarily be achieved by conforming to this requirement during model partitioning.

Solution(s) for Observation 11: Ensure that the partial model transformation tasks to not produce overlapping output artifacts. This eliminates the need to copy different output folders and to merge files.

B. Incremental Model Transformation

The incremental model transformation scenario starts somewhat different compared to the initial scenario. There are two versions of the source model, namely the previous model which is already transformed into an application, and the newly created model. Model modifications are made incrementally in the same way that an application is incrementally updated according to the model changes. The two model versions are then passed to the model transformation phase F in Figure 5). In **Observation 13** we mentioned the inefficiency of an approach where the generator determined the delta given the two source models. We argue that it might not be needed to load two complete models to determine the delta. In case the developers have control over the modeling environment it would be beneficial to keep track of changes whilst they are made. This way the delta itself is directly created and does not have to be computed using the two model versions. We see from the experiments that it takes 0.5 second to load a small model (less than 30 elements), like a delta, whereas it takes about 20 seconds to load the largest tested model (more than 1000 elements). Moreover, the modeling environment has knowledge and access to the latest model version and can thus directly include other elements that are connected through dependency relationships. This of course works only for dependencies in the model and traceability for model transformations is still required. However, we were unable to propose a solution for situations where no access to the modeling tool is available.

Solution(s) for Observation 13: We propose a solution where only the delta model itself is passed the generator. Access is needed to the modeling tool to directly create a delta model when a change is made by the end user.

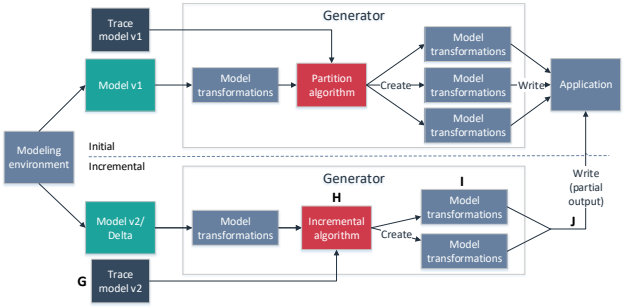

Fig. 5: Reference model evolution handling after the initial model transformation. An updated source model is passed to the generator which is loaded into memory. After that an updated version of the trace model must be calculated (G) on which the Incremental algorithm is dependent (H). Then the calculated delta model can be transformed by the generated incrementally and in possibly parallel depending on the overhead (I). Finally, a partial output is written to the already existing set of application files to update the application.

After the delta is loaded it must be transformed to update the application. Before this process can start additional calculations on the delta are needed to complement the delta with generation-time dependency elements. This is explained in detail in the Related work and Traceability case study sections. We group the logic required for these calculations in a new and separate module and place it into the model transformation pipeline. We call this module "Incremental algorithm" and can be found at letter I in Figure 5. This module requires a delta model and a traceability model. A traceability model is created in the same way as for the initial model transformation described earlier where it is used for model partitioning. For the simulation approach the new source model is used since the delta cannot be used to create a complete trace model.

Again, the last phase of the generator is to write the created artifacts to the application folder (J in Figure 5). Since at this point only a partial output is created the new and existing artifacts have to be merged in some way. As already described earlier file locking conflicts could occur when naively merging the partial out with existing artifacts. This problem can be solved by the suggestion provided for **Observation 11** above.

C. Open Concerns Regarding Traceability

While researching traceability we made additional observations that do not influence the design of the reference architecture. These observations are of a more practical nature and are therefore also important to take into account. First, inserted trace generation code must be kept up-to-date otherwise it cannot be used reliably (**Observation 2**). An IDE that can

support the developer in this task but we could not find such an IDE or any other mechanism to aid the developer in keeping the code snippets up-to-date. **Observation 3** argues that only relevant traces have to be collected. We consider it the responsibility of the developer to determine what is relevant what is not. However, no structured protocol or guidelines were found that could aid the developer in this process. This can reduce the number of traces collected which increase model transformation performance as is discussed in **Observation 6**. However, no structured method to do this is found in literature because of the many implementation details and possibly language specifics. Lastly, a suitable format in which the trace model can be stored is researched and analyzed in **Observation 4**. We could not compare both XMI and JSON formats in our case study generator and can therefore not provide a best practice or solution and depends also on the transformation platform used.

VII. Conclusions and Future Work

This section concludes the research by answering the designed research questions and presents our contribution to research on improving the performance in MDD. A related work study and analysis is performed to explore the available methods for improving the performance of a MDD generator (**SRQ1**). The domain of Model-Driven Development exist already for a long time and have matured over the years. Multiple transformation languages and tools are proposed where ATL is the most mature. All researched approaches aimed at improving the performance use ATL as the transformation language and leverage the underlying virtual machine for optimizations. The extensive focus on ATL is a drawback since the case organization use a different transformation language, i.e. a general-purpose programming language. This has significantly different properties compared to ATL that is a Domain Specific Language regarding model transformation. There is only little applicable literature available to our context which limits this research in the number of approaches that can be researched.

A common approach to gain insight into the transformations of a MDD generator is traceability (**SRQ2**). Traces are collected from the model transformations in the MDD generator and form a trace model. Traces are dependencies between two elements that take part in a model transformation. By analyzing the trace model the MDD generator can be visualized in terms of model dependencies. Moreover, it is used to reason about how a model is transformed into an output application. We consider traceability as a prerequisite for both model partitioning and incremental model transformation. Our case study shows that it is complex to collect all traces needed to use it effectively for model partitioning of incremental model transformation. We observed that it is time-consuming to modify our case generator due to the general-purpose language used for the model transformations. Since this approach is very expensive or even unfeasible another approach is briefly researched, namely abstract interpretation.

The insights provided by the trace model are utilized in the generator by partitioning the source model into one or more partitions (**SRQ3**). This can be achieved by analyzing the dependencies captured in the trace model and enables partial - and incremental model transformation. We researched a data parallelism approach which provides all components needed for partial model transformation. A valid partition can only be created when the trace model contains all possible dependencies given a source model. Valid here means that the partition can be generated without additional knowledge of other elements or dependencies. By implementing the data parallelism approach as part of a case study observations could be made which provide insights on what properties and processes are needed to improve transformation performance by using partial model transformation.

Finally, parallel and incremental model transformation are researched in a case study as two methods to improve the performance. Multiple partial model transformation tasks are combined to research parallel model transformation. An experiment is performed to prove that the implemented parallelization technique does indeed work and yield a performance improvement compared to sequential model transformation (**SRQ4**). Our parallel transformation reduce the total generation duration by 2.5 to 3.5 times which corresponds to experiment performed with ATL on MapReduce. A limitation of our approach is that the model partitioning is not based on traces since no complete trace model could be created. Instead, a certain model element type is used and the explicit dependencies from the source model are used. This approach does not work for a source model with more complex relationship where dependency information from the model transformations are needed to be able to generate a correct output.

The presented reference architecture and the corresponding design decision, protocols and practical take-aways answer the main research question: *How can the performance of model transformations be improved in Model-Driven Development?*. The reference architecture and corresponding documentation combined present a novel artifact and is designed to aid organizations in improving their MDD generator. We proved and argued that both parallel - and incremental model transformation, respectively, improve the performance of model transformations in MDD. The reference architecture is based on observations made during the case studies and single experiment and are to be extended and validated by performing additional case studies.

Acknowledgments

The authors would like to thank ..... for the fruitful discussions and feedback.

This research was supported by the NWO AMUSE project (628.006.001): a collaboration between Vrije Universiteit Amsterdam, Utrecht University, and AFAS Software in the Netherlands. The NEXT Platform is developed and maintained by AFAS Software.

REFERENCES

[1] A. Benelallam, A. Gómez, and M. Tisi, "ATL-MR: Model Transformation on MapReduce," *Proceedings of the 2nd International Workshop on Software Engineering for Parallel Systems*, pp. 45–49, 2015.

[2] A. Benelallam, A. Gómez, M. Tisi, and J. Cabot, "Distributed Model-to-Model Transformation with ATL on MapReduce," *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering*, pp. 37–48, 2015.

[3] A. Benelallam, M. Tisi, J. Sánchez Cuadrado, J. De Lara, and J. Cabot Ef, "Efficient Model Partitioning for Distributed Model Transformations- ficient Model Partitioning for Distributed Model Transformations," *ACM SIGPLAN*, 2016.

[4] A. Bergmann, G., Horváth, Á., Ráth, I., Varró, D., Balogh, A., Balogh, Z., & Ökrös, "Model Driven Engineering Languages and Systems," *Model Driven Engineering Languages and Systems*, pp. 76–90, 2010.

[5] J. Bézivin, G. Dupé, F. Jouault, G. Pitette, and J. E. Rougui, "First experiments with the ATL model transformation language: Transforming XSLT into XQuery," *2nd OOPSLA Workshop on Generative Techniques in the context of Model Driven Architecture*, vol. 37, 2003.

[6] J. J. Dongarra and D. C. Sorensen, "A portable environment for developing parallel FORTRAN programs *," *Parallel Computing*, vol. 5, pp. 175–186, 1987.

[7] M. Huchard, C. Nebut, M. Transformations, T. Neple, and J. O. Ecmda, "Towards a Traceability Framework for Model Transformations in Kermeta Towards a traceability framework for model," 2006.

[8] H. Hussmann, G. Meixner, and D. Zuehlke, *Model-Driven Development of Advanced User Interfaces*, 2011.

[9] V. Lussenburg, T. van der Storm, J. Vinju, and J. Warmer, "Mod4J: a qualitative case study of model-driven software development," *Model Driven Engineering Languages and Systems*, pp. 346–360, 2010.

[10] K. Muthukumar and M. Hermenegildo, "Compile-time derivation of variable dependency using abstract interpretation," *The Journal of Logic Programming*, vol. 13, no. 2-3, pp. 315–347, jul 1992.

[11] B. Selic, "The pragmatics of Model-Driven Development," *IEEE Software*, vol. 20, no. 5, pp. 15–25, 2003.

[12] J. Subhlok, J. M. Stichnoth, D. R. O 'hallaron, and T. Gross, "Exploiting Task and Data Parallelism on a Multicomputer," *Fourth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, 1993.

[13] M. Tisi, S. Martinez, and H. Choura, "Parallel Execution of ATL Transformation Rules," *Parallel Execution of ATL Transformation Rules. MoDELS*, pp. 656–672, 2013.

[14] D. Varró, "Patterns and styles for incremental model transformations," *PAME@ STAF*, pp. 41–43, 2015.

[15] H. P. Zima, H.-J. Bast, and M. Gerndt, "SUPERB: A tool for semi-automatic MIMD/SIMD parallelization *," *Parallel Computing*, vol. 6, pp. 1–18, 1988.