Performance Prediction Upon Toolchain Migration in Model-Based Software

by

Mohamed Aymen Ketata

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2015

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Changing the development environment can have severe impacts on the system behavior such as the execution-time performance. Since it can be costly to migrate a software application, engineers would like to predict the performance parameters of the application under the new environment with as little effort as possible.

In this work, we concentrate on model-driven development and provide a methodology to estimate the execution-time performance of application models under different toolchains. Our approach has low cost compared to the migration effort of an entire application. As part of the approach, we provide methods for characterizing model-driven applications, an algorithm for generating application-specific microbenchmarks, and results on using different methods for estimating the performance. In the work, we focus on SCADE as the development toolchain and use a Cruise Control and a Water Level application as case studies to confirm the technical feasibility and viability of our technique.

Acknowledgements

I would like to thank my supervisor Prof. Sebastian Fischmeister for his continuous guidance, insights and support throughout my master. Thank you for giving me the opportunity to work flexibly and for always finding time for me. Your collaboration made the achievement of this work possible.

I would also like to thank Carlos Moreno for his fruitful discussions, continuous assistance, and advice. Thank you for being patient and motivating my research.

I would like to thank Prof. Krzysztof Czarnecki and Prof. Werner M. Dietl for reading and reviewing my thesis.

For the discussions on constraint solvers and model-based applications, and reviewing my work, I would like to thank Jimmy Liang.

To my family, thanks for your love and support all these years.

To my friends, thanks for making my studies at Waterloo full of joyful moments and great souvenirs.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Model Driven Development (MDD) is a software development approach where the source code can be automatically generated from the models. MDD is used to build an abstract representation of the system to improve productivity and communication among the managers, architects, designers, and developers. Some modeling languages such as Unified Modeling Language (UML), SysML, and modeling tools such as Eclipse modeling framework plugins are used in MDD development [30]. MDD provides several improvements to the development process of software. MDD reduces, for example, the cost of failure due to technical implementation problems. The modeling software tool checks the errors and produces valid models. MDD reduces the gap between domain experts and developers as the models check for the compliance with the requirements. Changes can be easily made to incorporate the business feedback into the models. The models are more intuitive than the source code, and hide the technical challenges of the implementation [20].

MDD requires specific toolchains to transform abstract models into executable code for the target system. A toolchain is a set of software tools used to generate code, compile and link, and it provides the binary code. In the context of real-time safety-critical applications, a qualified toolchain assures that the assertions and requirements claimed at the top-level are valid at the target programming language level.

Developers are reluctant to upgrade the toolchain while developing application models because, among other things, even a small change in the toolchain can incur a migration cost and can add complexity in building the program. Changes in the toolchain can significantly affect the system behavior such as the execution performance of the program created with one toolchain. Various factors might motivate the migration decision to a new toolchain. For example, the new toolchain might offer new features that we want to benefit

from, or the old toolchain might lack support and maintenance. We focus in this work on the performance parameters as the major decision factor of the migration. Predicting the performance parameters, before migrating the model to the new toolchain, can significantly help making the upgrade decision. Porting the application to a new toolchain should reflect a deep understanding of the performance changes under the new toolchain. The lack of structured and effective techniques to migrate the application to the new toolchain may lead to engineering work and an expensive migration with uncertain outcomes. This issue is critical in the embedded systems as the use of a new toolchain may also add safety and security issues. The upgrade of a verified toolchain such as the SCADE Systems is more difficult due to the strict requirements of the generated binary code [2, 25].

We provide a framework to predict the execution-time performance of a model-driven application on a new toolchain without migrating the entire application from the original toolchain. As the migration cost is in principle proportional to the amount of engineering work, our framework provides an automation process to extract and generate application-specific microbenchmarks. The execution-time estimates are relevant for the toolchain upgrade decision due to the trade-off between the cost and performance benefits. We present an automated technique to efficiently analyze the application model with respect to the new toolchain before the migration process as we can extrapolate and produce an accurate prediction of the execution-time performance.

The remainder of this thesis proceeds as follows: Chapter 2 presents the problem statement and outlines our approach. Chapter 3 defines the used technical terms. In Chapter 4, we discuss several studies presenting ideas or techniques related to our work. Chapter 5 provides an overview of our developed tool. Finally, we describe our experimental setup in Chapter 6 and results in Chapter 7. We discuss the validity and usability of our framework in Chapter 8, followed by concluding remarks in Chapter 9.

Chapter 2

Overview

This chapter introduces the problem that our work addresses and provides an overview of our proposed approach. Figure 2.1 illustrates the problem statement.

2.1 Problem Statement

Given two toolchains $\mathcal{T}_1$ and $\mathcal{T}_2$ and assuming a model has been developed and compiled to an executable under $\mathcal{T}_1$, predict, with minimal porting effort expressed as the number of changes in the model, the execution-time performance, in the same execution environment, of the executable generated using $\mathcal{T}_2$.

Given a performance prediction technique and a set of metrics, develop a systematic method to compare the metrics and identify the optimal one for the application domain.

We refer to the solid lines path in Figure 2.1 as the anticlockwise path or inexpensive path. It represents the workflow of our approach to estimating the performance parameters. It is based on the extraction of metrics from the application model in $\mathcal{T}_1$, and the generation of application-specific microbenchmarks in $\mathcal{T}_1$. The next step is the migration of the microbenchmarks from $\mathcal{T}_1$ to $\mathcal{T}_2$. We explain the details of this path in Chapter 5. We refer to the dashed lines path as the clockwise path or expensive path. It represents the steps to follow if the entire application was migrated from $\mathcal{T}_1$ to $\mathcal{T}_2$, which is in principle what we try to avoid for the purpose of performance prediction.

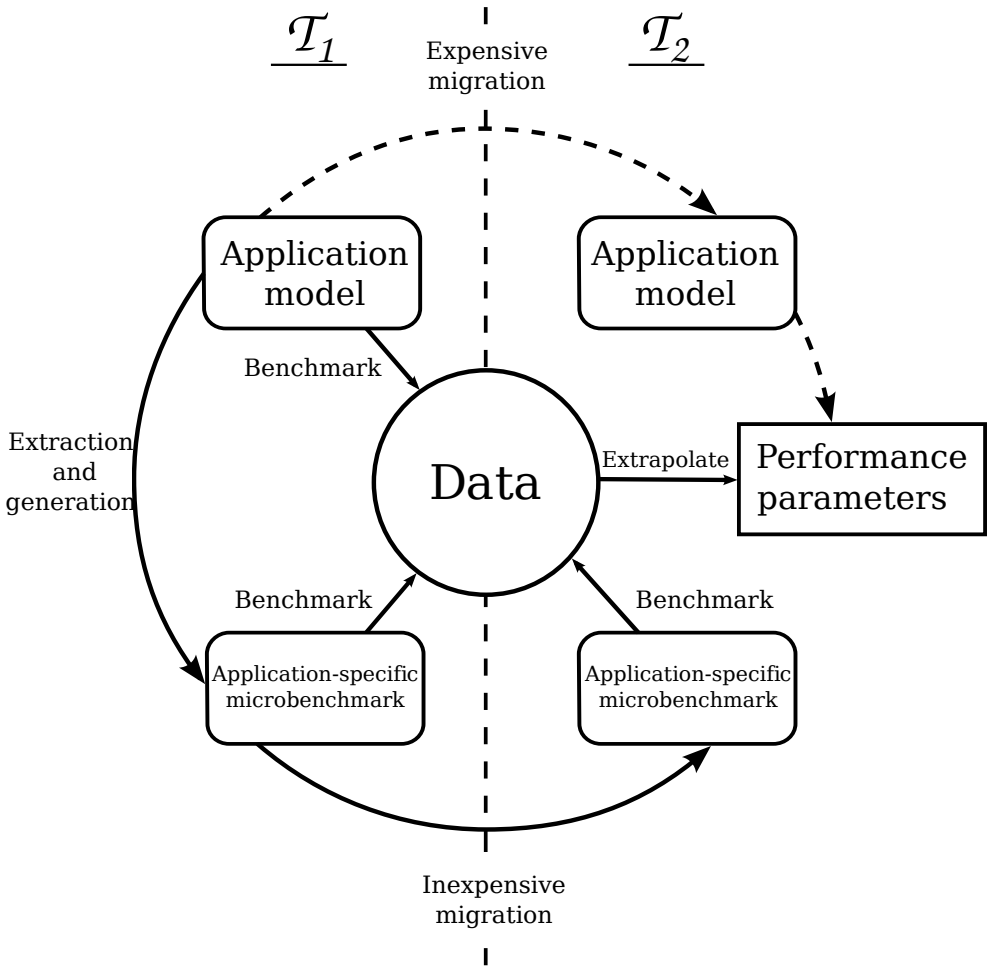

Figure 2.1: Description of the problem statement

2.2 Our Approach

Our proposed approach allows developers to avoid porting the entire application when they face the decision of upgrading a development toolchain. Instead, we can predict, through benchmarking analysis, the performance of the application under the new toolchain $\mathcal{T}_2$. This estimation is done at an early stage and avoids the cost of the migration of the entire application. The software artifacts such as the requirement, the specification, and the design documents that describe the abstraction of the final software system should be used to understand the application characteristics. To this end, we need to define the

properties of the application under toolchain $\mathcal{T}_1$ that have an impact on the performance and derive application-specific microbenchmarks. To generate microbenchmarks that are representative of the application model, we use software metrics that capture the features and patterns in the application model that are relevant to its performance. Our approach focuses on the performance of model-based application relative to a toolchain rather than the application's architecture or design.

Our approach is similar to a Monte Carlo simulation [38], in fact, we generate random samples (application-specific microbenchmarks) to predict performance parameters. The accuracy of the prediction is based on the quality of the samples to capture the performance characteristics of the application. The purpose of using software metrics is to capture these properties and generate representative samples.

Our framework follows the inexpensive path from Figure 2.1. The framework is based on a set of tools that can automatically analyze the application, identify relevant characteristics in the model, and generate microbenchmarks that are representative of the original model. Users of the framework need to follow these steps:

1. **Measure application model with characteristic metrics:** The first step is to take the application model and characterize the model using a set of metrics. In our work we only use the ratio and network centrality metrics, however, other metrics such as Cyclomatic complexity, Helastead complexity, and Fan In/Fan-out [26, 37] are also applicable.

2. **Encode measurements in constraints:** The next step is to use the measurements and set up a series of constraints. These constraints will ensure that generated microbenchmarks are representative of the application model and maintain the original performance-relevant properties.

3. **Generate application-specific microbenchmarks:** The constraints permit multiple solutions. Depending on the required precision of the prediction, users then generate microbenchmark models from solutions to the constraint set. Being a statistical process, we can expect that increasing the number of microbenchmark models will lead to a lower standard error in the prediction

4. **Port microbenchmarks and generate code:** The migration of the microbenchmarks to the new toolchain should be relatively straightforward compared to the migration of the entire application model. Consequently, the porting effort will be inexpensive with respect to the required engineering effort. After porting, the user will compile the microbenchmarks on both toolchains and prepare the microbenchmarks for execution on the target platform.

5. **Benchmark microbenchmarks on the target platform:** By benchmarking the different models under the two toolchains and analyzing the results, the user can extrapolate an estimate of the execution-time performance of the entire application as migrated under the new toolchain.

Case studies of the SCADE toolchain for MDD provide evidence that our approach and framework are feasible. We used the Cruise Control and Water Level applications developed under two versions of SCADE systems and compared the predicted results to the migrated ones. Our estimates were reasonably accurate and correctly predicted that the executions of the applications under SCADE 6 were faster with respect to the applications under SCADE 5.

We remark that our approach does not predict the Worst-Case Execution Time (WCET). The presented method and tool constitute decision support to lower the risk when considering switching between toolchains. They focus on predicting average expected performance parameters while migrating toolchains and not on extremes like the WCET, which requires additional analysis and optimization. WCET analysis is required once the entire model is migrated to a different toolchain. For safety-critical systems, regulations most likely will require that a new analysis be made upon any change in the system, regardless of any estimates that would have been made prior to the changes.

Chapter 3

Background and Terminology

The SCADE Suite is a model-based development environment specifically tailored for safety-critical systems and often used in the avionics domain. The SCADE Suite is an integrated development environment that includes model-based design, simulation, verification, and qualified code generation. As SCADE provides a synchronous approach for reactive programs, it is suitable for developing safety-critical embedded software such as automotive and avionics applications [15].

Estimation or *Parameter Estimation* is the process of obtaining an approximate value of a parameter given available, and in general insufficient, data. In statistics, estimation usually refers to finding a function f of an observation vector X such that the error $|f(\mathbf{X}) - \theta|$ when estimating the parameter θ given an observation $\mathbf{X}$ is minimized in some sense [39].

Information Extraction (IE) recognizes entities and relations from sets of data and transforms them into structured representations. In this work, we focus on the extraction techniques used to generate application-specific microbenchmarks from the application models.

Migration is the conversion of a model developed under one version of a toolchain to either a new version of the same toolchain or a different toolchain. Unit, regression, and integration tests should be performed to validate the migration and ensure the required quality. The migration can affect the application behavior such as the performance parameters. The migration may be time-consuming, costly, and hard to perform manually. The automation of the migration process could be used to reduce cost, but there are still important challenges to migrate concrete models [35].

Application Model refers to a set of large and complex models developed for a specific domain such as aircraft engines. Building such an application model requires the

integration of physical, mathematical, and computational models.

An *application-specific microbenchmark* or *fingerprint model* is a reduced size model that shares common characteristics with the entire application model. Despite the fact that the microbenchmarks are randomly generated, they are representative of the application as they are constrained to a specified set of metrics extracted from the application.

Ratio metric is a software metric that consists of the fractions of each type of blocks in a model. That is, for each type of block, the number of blocks of that type divided by the total number of blocks is associated with the type. This metric does not consider I/O connections or the specifics of the design and structure of the model. Given a model, the computational cost of extraction of the ratio metric is linear with the number of blocks. We present a more detailed discussion and intuition on why this metric is relevant to the performance analysis in Section 6.3.

Chapter 4

Related work

The notion of software metrics is one of the key aspects in our method, as it is what captures the characteristics of a model that are relevant to its performance. Several studies exists in the literature that deal with this idea in the context of programming languages [10,18,19]. These software metrics include: Lines of Code, Cyclomatic complexity, Healstead complexity, Cohesion and Coupling, Fan-In/Fan-Out and NPath. The Cyclomatic complexity [37] indicates the complexity of the application. It is computed based on the independent paths in the application generated with conditional statements. NPath complexity [13] measures the number of possible outcomes from the application. It might be hard to compute for large models that have nested conditional blocks. Fan-In/Fan-out [26] focuses on the information flow and measures the connections among the application components. Cohesion refers to the module responsibility and functionality as it expresses the degree of interdependency between the elements of the module. Coupling refers to the degree of dependency between the application modules. It explains the strength of the connection between the modules. Though these studies look for metrics that describe and capture the important characteristics of an application, like our work, they focus on program maintainability rather than performance and thus are not directly relevant to the problem that we are addressing.

We reviewed the applicability of software metrics to SCADE models. The work in [32] implemented a SCADE metric interpreter framework to extract the characteristics of the SCADE models. Some other related software metrics such as controllability and observability were presented in [16]. These metrics analyze the testability of the SCADE programs. Testability metrics try to identify the different parts of the application that are critical, prone to errors, and difficult to validate. Similar ideas have also been investigated in the

context of migrating legacy applications to modern programming languages with a focus on the quality control of the application [34].

The survey [4] reviews research work that focuses on the performance prediction of model-based applications. The proposed approaches address the integration of performance analysis at early stages of the development process. Several works [23, 41] review model-based performance prediction approaches. These studies focus on the importance of conducting performance analysis at different stages throughout the software development cycle. Performance prediction requires a deep analysis of the system architecture from the requirement and specification phase to the configuration and deployment phase. Several works explore the performance requirements and try to predict the performance parameters at early stages to identify the issues of the concrete integration [17]. Some approaches, such as [27], focus on the application behavior after changes in source code and its impact on the application environment, dependencies, and performance. By contrast, our approach focuses on the changes introduced by the migration to a new toolchain.

Liu et al. [28] present an analytical approach that relies on stochastic modeling to predict the performance parameters of component-based applications. Our approach is statistical and based on measurement data from the actual target platform. The tool SoftArch/MTE [24] focuses on the evaluation of test-beds generated under various architectures of the application to help choose a particular architecture for the application design. In contrast, we estimate the effect of the migration between toolchains for the same application.

The use of an approach based on metrics for model-based applications was presented in [29] to analyze the application and detect failures. Menkhaus et al. introduced the network metrics (betweenness and closeness) in model-based developed application to investigate the stability of the application and its vulnerability to failure [29].

The approach of converting a model-based application to its graph representation is a pre-processing step that is widely used to solve research problems. For example, Deissenboeck et al. used in [14] the graph representation to detect models clone of Simulink models, while Agrawal et al. investigate in [1] the possibility of formally validating the models by visualizing the graph representation.

Chapter 5

Methods and Tools

We now present the implementation of the framework and the methods and tools supporting it. To be able to evaluate the feasibility and viability of our concepts, we developed a tool that can generate application-specific microbenchmarks from a SCADE application model. Figure 5.1 gives an overview of the process. We detail in this chapter the implementation of the workflow presented in Figure 5.1. We start by extracting the metrics from the application models. We present in Section 5.1 the metrics used to analyze the application and identify its characteristics, and the techniques used to extract them. We then describe the process of encoding these metrics as constraints in the Clafer modeling language and the use of a constraint solver to generate application-specific microbenchmarks.

5.1 Metrics

The metric choice is relevant to our study as it is the key factor to generate representative microbenchmarks. We investigated several software metrics that are relevant to our study. We present in this section the ratio and network metrics.

5.1.1 Ratio Metrics

The ratio metrics are intuitive metrics that captures the total number of blocks and their data type. These metrics are intuitive and can be easily extracted.

- **Static ratio** is the fractions of each type of blocks in a model. Given a model, the extraction of this metric is linear to the number of blocks in the model. There are

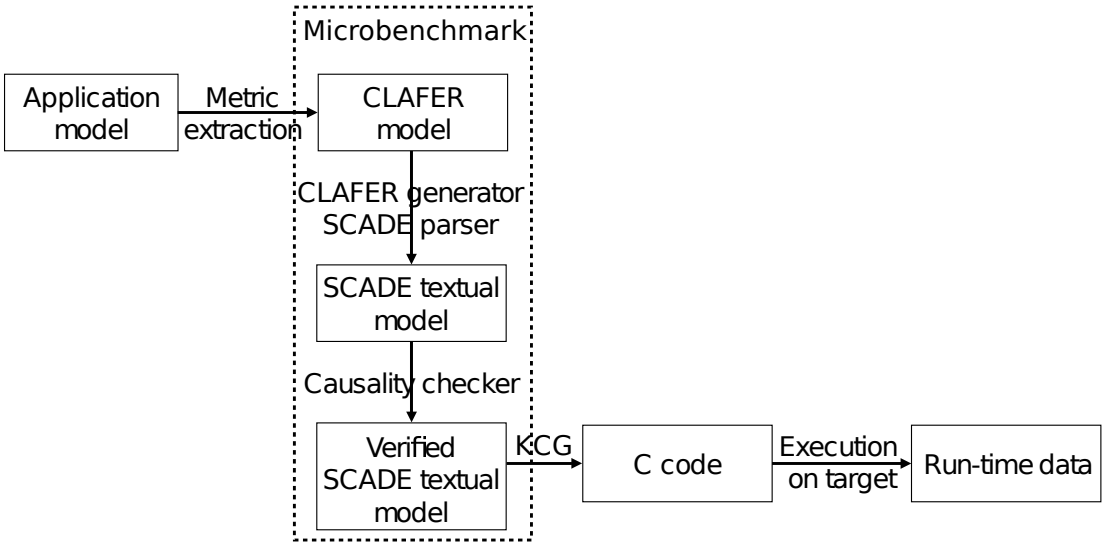

Figure 5.1: Workflow generation

two methods to extract the static ratio for block-based languages. The static ratio can be extracted from the SCADE model representation or its generated C code. Each predefined block has a unique ID that identifies it. We used Tool Command Language (TCL) scripts to navigate through the SCADE models and counts the number of occurrence of each type of blocks in the model. For the second method of extraction, we developed a script that parses the C code and counts the occurrence of the predefined SCADE blocks. The script analyzes the C code without running it.

- **Dynamic ratio** is the fractions of each type of blocks called during the execution of the application model. We injected counters for each type of block into the C code. Each counter is incremented every time its associated block is triggered. The dynamic ratio captures only the executed SCADE blocks and ignores the non-executed ones. To statistically execute all the possible application paths, we uniformly sampled pseudo-random values as inputs across the application. However, we noticed that SCADE 5 executed all paths of the application and the numbers of the counters introduced by the dynamic ratio always produced the same numbers. We adjusted the dynamic ratio for the SCADE 6 behavior.

  Following, we present the entire process to extract the dynamic ratios:

1. Inject counters into the C code of the SCADE 5 application model.
2. Run the SCADE 5 application, extract the counters and compute the dynamic ratio.
3. Generate SCADE 5 microbenchmarks and "temporary" SCADE 6 microbenchmarks.
4. Inject counters into the "temporary" SCADE 6 microbenchmarks.
5. Run the "temporary" SCADE 6 microbenchmarks n times with random inputs.
6. Extract new counters values from SCADE 6 microbenchmarks.
7. Extract the ratio from the new dynamic counters of all microbenchmarks.
8. Generate new SCADE 6 microbenchmarks.
9. Benchmark the SCADE 6 microbenchmarks generated at Step 8 and the SCADE 5 microbenchmarks generated at Step 3.

5.1.2 Network Metrics

While the ratio metrics highlight the fractions of operators and the data type flow, the network centrality metrics emphasize the interaction of each individual block with other blocks and the impact of its operation on the computed data flow.

The network metrics aim to reflect the structure of the SCADE model by analyzing each SCADE operator and capturing its importance within its environment. The network metrics capture each block characteristics and its impact on the model structure. It is different from the ratio metric as it investigates the effect of each block within blocks from the same type. In other words, the ratio metric, for example, considers the Plus blocks as an entity while the network metrics investigate each Plus block apart. We investigated several metrics on the graph such as fan-in, fan-out, degree, closeness, betweenness, etc. Some metrics were not relevant to our study. We excluded, for example, the fan-in metric as the number of inputs for each category of blocks is fixed by the SCADE grammar constraints. For example, all math, comparison, and Boolean blocks have two inputs while conditional blocks have three inputs. In this sense, the fan-in metric fails to capture the difference between the various blocks and their types.

Building the Graph Representation

To use the network metrics on model-based applications, we generated the graphs corresponding to the models. SCADE is a modeling language that can be represented as a

graph. The SCADE blocks can be considered as nodes in the graph representation while the connections between the nodes can be considered as edges.

We developed TCL scripts that capture the expressions and equations of the application models. The scripts navigate through the application models and the library files. Based on the UML meta-models of the SCADE language, the scripts identify the class of the objects to report the inputs, constants, expressions and equations. The scripts can parse the classes added by the user such as redefined types, structures, and constants. By running the scripts, we can capture the entire structure of the models.

The graph representation provides an intuitive visualization of the model structure and the interaction between its blocks. Building the graph requires parsing the information reported by the TCL scripts. The graph generation tool parses the predefined SCADE blocks and the local variables to build a dictionary of the connections between the different blocks. The type of the SCADE blocks is captured based on the type of local variables, constants or inputs. Using the collected information, we can generate the nodes and the connections between them. The generated graph is a directed network. Figure 5.3 is an example of a generated graph, which is equivalent to the SCADE model of Figure 5.2.

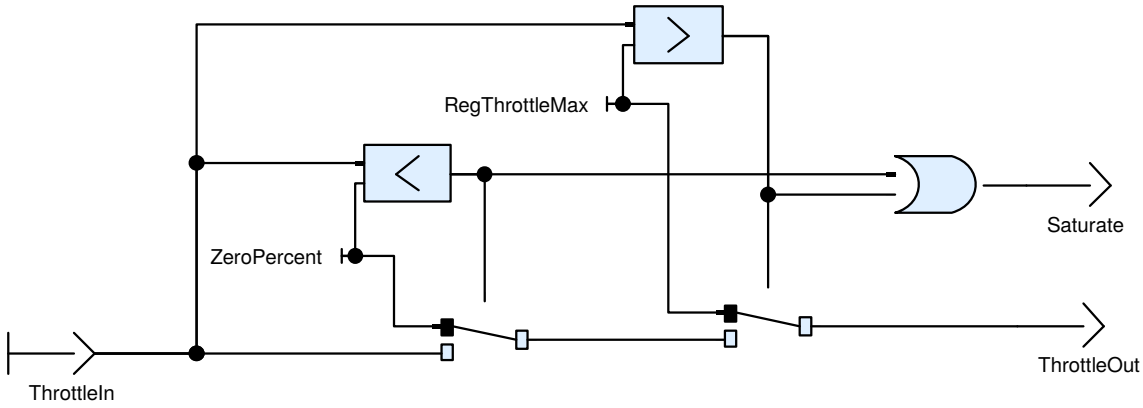

Figure 5.2: Saturate Throttle SCADE model

The graph generation tool explodes recursively the models called by other models to use only the primitive predefined SCADE blocks such as Plus, Minus, Multiply, AND, etc. Figure 5.7 shows an example of the graph of the SCADE model presented in Figure 5.4. Figure 5.7 is the graph of the Throttle Regulation model. It reflects the call to the Saturate Throttle model as it includes its graph presented in Figure 5.3. The tool generates the graphs in Graph Modeling Language (GML) and Graph Description Language (DOT) representations for usability purposes. The use of these formats made the generated graphs

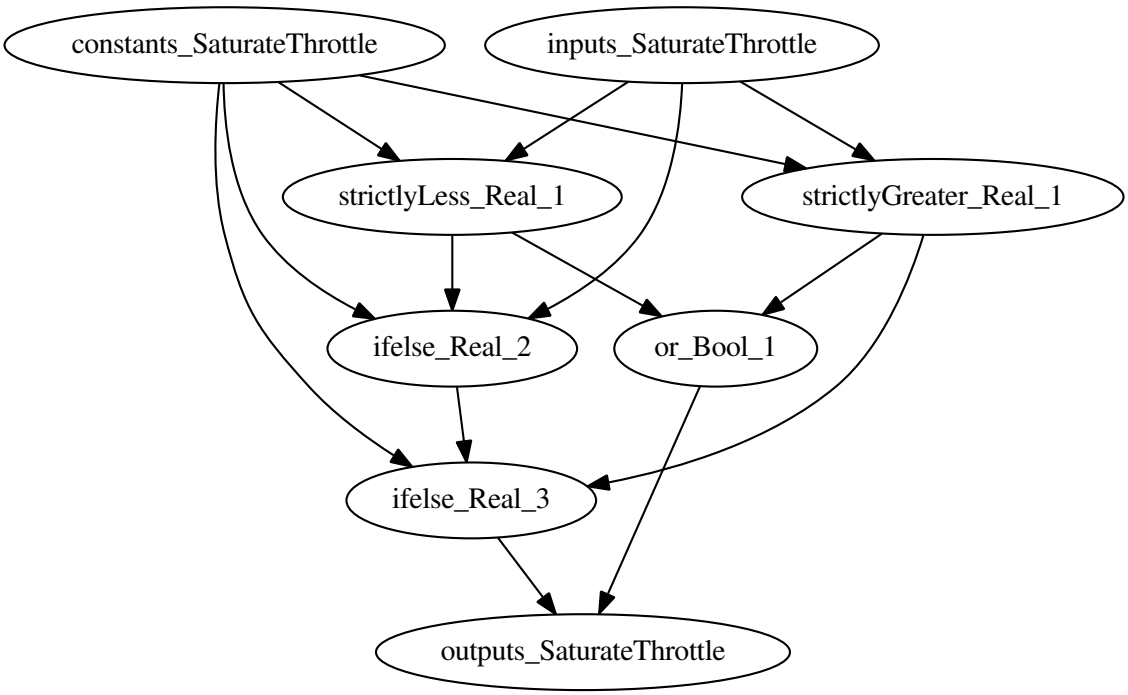

Figure 5.3: Saturate Throttle graph representation

compatible with various analysis and drawing tools. We present in Listing 2 the Saturate Throttle model in DOT representation.

The models presented in Figure 5.3 and Figure 5.4 are developed by ANSYS SCADE [40] and are provided as example models.

We focused on centrality metrics [22], which determines the importance of a node in the graph. The term importance is subject to various interpretations. We present the centrality metrics that aims to investigate the importance of the data flow of the SCADE blocks and the importance of the SCADE operators within the SCADE model. The centrality metrics assign a real value to the nodes, which reflects its importance weight. To extract the network metrics, we used the Python package Networkx, which provides several methods to construct, analyze and study complex networks.

- **Fan-out** of a node refers to the number of edges pointing out of the node [26]. The fan-out metric is important as it reflects the number of SCADE blocks and model outputs that depend on the data flow generated by the operator. The extraction of the fan-out metric is linear to the number of blocks.

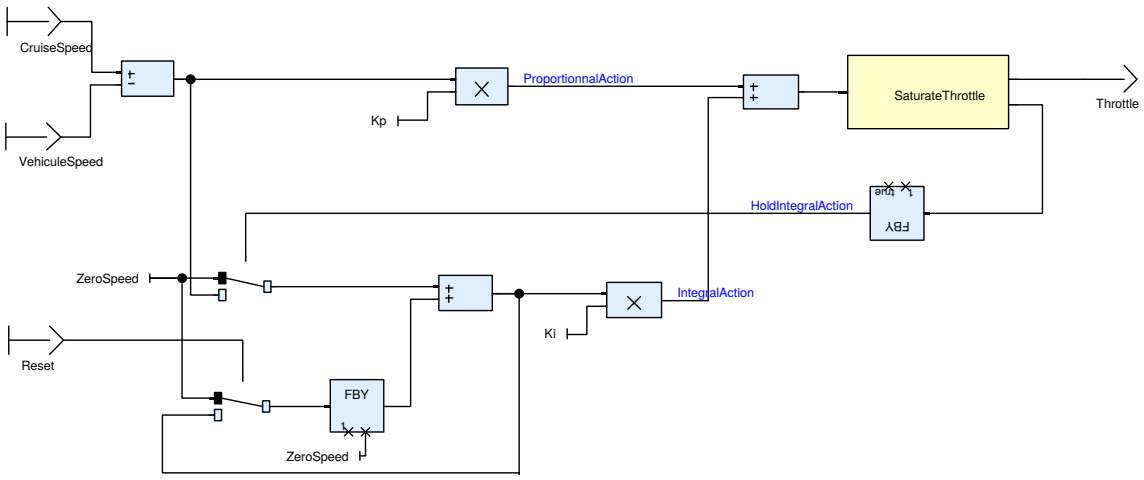

Figure 5.4: Throttle Regulation SCADE model

- **Page Rank** is a Google search algorithm used to rank web pages based on the count and the quality of its incoming links [33]. The algorithm assigns higher values to the most referenced pages. PageRank highlights the most referenced blocks in the model, the relevant ones of the model structure.

- **Closeness** of a node is the inverse of the sum of shortest path distances from a given node to the entire nodes in the network [31]. In our study, the closeness metric reflects the importance of a SCADE block to communicate the generated data flow to other SCADE blocks. A large closeness value for a particular node implies that the data flow would navigate a short distance to impact the model.

- **Betweennes** of a node measures the number of times the node acted as an intermediate in the shortest path between two other nodes [21]. In other words, given a node, betweenness reflects the number of pairs of nodes that has the selected node in the shortest path to reach one another. Betweenness reflects the strength of the connection between a SCADE block and the other blocks in the model.

- **Eigenvector** measures the influence of a node on the network [6]. It is based on the assumption that each node's centrality is the sum of the centrality values of the nodes that it is connected to. The eigenvector metric reflects the centrality of a given node with respect to all other nodes in the graph as it weights each connexion between the nodes.

Table 5.1 shows the result of the described network metrics for the Throttle Regulation

model presented in Figure 5.4

	Fan-Out	PageRank	Closeness	Betweenness	Eigenvector
ifelse_Real_7	2.857	6.123	15.870	27.924	23.577
ifelse_Real_2	2.857	4.700	12.895	6.579	23.511
ifelse_Real_3	2.857	6.570	13.940	16.959	28.698
inputs_ThrottleRegulation	8.571	0.967	21.126	0	0
minus_Real_2	5.714	1.516	22.088	3.801	0
fby_Bool_2	2.857	5.114	14.529	26.608	30.155
outputs_ThrottleRegulation	0	5.114	0	0	30.155
inputs_SaturateThrottle	8.571	7.502	17.484	35.965	16.876
ifelse_Real_8	2.857	6.404	12.429	4.971	27.607
constants_ThrottleRegulation	14.286	0.967	24.951	0	0
or_Bool_1	2.857	3.771	13.940	13.450	20.632
plus_Real_3	5.714	11.761	17.484	35.088	35.310
plus_Real_2	2.857	7.688	15.870	37.135	21.585
multi_Real_1	2.857	1.776	14.803	7.310	0
multi_Real_2	2.857	6.130	14.529	26.608	27.607
strictlyLess_Real_1	5.714	3.298	16.118	6.579	13.195
fby_Real_3	2.857	6.576	14.529	4.240	21.585
strictlyGreater_Real_1	5.714	3.298	14.529	12.865	13.195
outputs_SaturateThrottle	5.714	9.757	15.170	33.333	38.569
constants_SaturateThrottle	11.429	0.967	20.776	0	0

Table 5.1: Network metrics of the Throttle Regulation model

After extracting the metrics, we encode them as constraints. We focus on the static ratio metric as an example to detail the process of generating microbenchmarks as shown in the workflow of Figure 5.1.

Figure 5.5 is an example of a SCADE model. It has six math operators, one comparison operator, and one Boolean operator. Based on the presented metrics, we encode the metamodels of the modeling language and the metric extracted values as a set of constraints and feed them into a solver. We use Clafer [3] as the constraint language and its associated solver, which is based on the Choco constraint solver. Table 5.2 shows block occurrence and ratio data for the model in Figure 5.5.

Given the extracted measurements in Table 5.2, we produce constraints for the solver

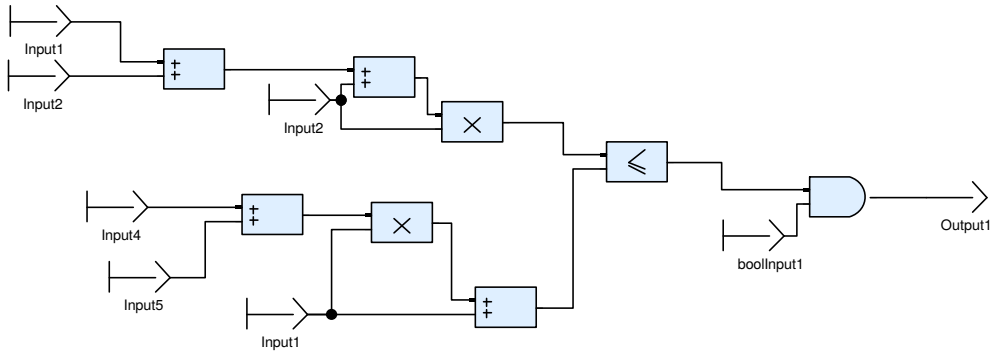

Figure 5.5: SCADE model example

tool. For example, given that the size of the microbenchmarks is n, the number of Plus operators would be $50\% * n$, the number of Multiply operators would be $25\% * n$, and the number of And operators would be $12.5\% * n$

	Occurrence	Ratio
Plus	4	50%
Multiply	2	25%
LessOrEqual	1	12.5%
And	1	12.5%

Table 5.2: Extracted measurements data

Clafer (**cla**ss, **fe**ature, **r**eference) is a lightweight modeling language with first-order relational logic. The Clafer compiler takes a model written in the Clafer modeling language as input and does some processing before invoking a backend solver to output instances conforming to the input model. Multiple solvers are supported such as Boolean SAtisfiability (SAT), Satisfiability Modulo Theories (SMT), and Constraint Satisfaction Problem (CSP). In this work, the input is a SCADE metamodel written in Clafer, conjointed with the metric data constraints, and the output is a random SCADE model generated by the CSP solver.

Listing 5.1 shows an example of the Clafer representation of the constraint: we want to generate exactly one addition operator either integer or real. We only show the constraints in Listing 5.1; while a minimized description of the SCADE modeling language can be found in Listing 1. Line 1 in Listing 5.1 specifies that plus_Int is an instance of the class MathBlockInt. The cardinality constraint `0..n` indicates that we would like to generate

between 0 and n instances. Line 3 represents the constraint the sum of the number of instances of plus_Int and the number of instances of plus_Real must be equal to 1.

```
1 plus_Int  : MathBlockInt  0..1
2 plus_Real : MathBlockReal 0..1
3 [#plus_Int + #plus_Real=1]
```

Listing 5.1: Metric data encoded in Clafer

Clafer generates random solutions, subject to the metric data and SCADE metamodel constraints, which we parse and convert to textual SCADE language. Textual SCADE is a declarative language; each line defines an element (e.g., one connection) in the model structure. Thus, the order of the lines is usually not relevant as in the case of imperative languages such as C.

For each set of constraints, we generated multiple random microbenchmarks. This is necessary to obtain a statistical characterization of the microbenchmark models' performance, which is necessary for the prediction. Listing 5.2 shows an example of the textual representation of one generated microbenchmark in the SCADE language. In this example, the first line indicates that we have an addition between the inputs intInput1 and intInput2, and the result is stored in the variable plus_Int0.

```
1  function microbenchmark( /* Inputs , Outputs */)
2  var
3   /* Local variables */
4  let
5   /* Update inputs */
6
7  plus_Int0 =  intInput1 + intInput2;
8  plus_Int1 =  plus_int0 + intInput3;
9  and_Bool0 =  boolInput2 and lessOrEqual_Int0;
10 lessOrEqual_Int0 =  multi_Int0 <= plus_Int0;
11 multi_Int0 = plus_Int1 * intInput4;
12 multi_Int1 = intInput5 * plus_Int0;
13  /* Update outputs */
14 tel
```

Listing 5.2: SCADE code of a sample microbenchmark

We generated C code using the toolchains KCG5.1 and KCG6.4 to compare the performance of the two toolchains. KCG is the automated code generator tool used to generate C code from SCADE models. KCG5.1 is the code generator tool used for SCADE 5 models, and KCG6.4 is the code generator tool used for SCADE 6 models. We automated the entire process flow to run the experiments. The scripts encode the metric, extract the

application characteristics, generate SCADE microbenchmarks, run the SCADE checker, fix any causality errors by adding delay blocks, and generate C code. We compiled and executed the generated code for benchmarking. The DataMill infrastructure was used in our benchmarks to evaluate the performance of the C code generator of the SCADE toolchain. DataMill offers various architectures and software and hardware factors that can be used in the performance evaluation [12]. We used the x86_64, i686, and ARM architectures to benchmark the C code, and we used the same hardware and software factors for the benchmarks.

5.2 The Constraint Solver

CSP [36] is a class of problems originating from the artificial intelligence community. A CSP problem is conventionally specified as a triple: V the set of variables, D the set of the variables' domains, C the set of constraints. A solution to a CSP problem is an assignment for each variable to a value in its domain such that none of the constraints are violated.

A CSP solver searches for solutions of a CSP problem by constructing an implicit search tree, where the variables are vertices and edges are assignments. The solver traverses the search tree in pre-order looking for leaf vertices such that every variable is assigned to a value, and none of the constraints are violated. These leaves are the solutions. The CSP backend for Clafer is implemented with the Choco library which supports integer variables and set variables over integers. Set variables are necessary and sufficient to encode the relational semantics of Clafer. For performance reasons however, the backend will optimize by using integer variables in place of set variables whenever possible. Choco's solving algorithm is based on an implicit search tree. The tree traversal can be broadly described in three steps starting at the root node:

1. Variable selection: Pick an unassigned variable using a heuristic. Most illustrations of CSP search trees would label the current node with the picked variable. Suppose the heuristic picked the integer variable i with domain $\{0, 1, 4\}$.

2. Decision: Assign the picked variable to a value in its domain and move down the current node's left branch. The right branch corresponds to the negative decision for when the algorithm *backtracks* to this node in the future. For example, if the left branch is $i = 1$ then the right branch is $i \neq 1$. More specifically, the left (respectively right) branch sets the domain of variable i to $\{1\}$ (respectively $\{0, 4\}$). There are other ways of making decisions such as domain splitting.

3. Constraint propagation: Infer new domains based on the available constraints. For example, if $i \neq j$ is a constraint, then remove 1 from the domain of variable j because assigning j to 1 will violate the constraint. If 1 is the only value in the domain of j, then the search entered a *contradiction* and can no longer proceed because the constraint $i \neq j$ is violated. The search then backtracks up the left branch(es) and goes down the nearest right branch and proceeds from there. Goto step 1 for the current node and repeat.

To generate random solutions, we modified the underlying CSP solver to construct the search tree randomly, i.e. the vertices/variables and edges/assignments are chosen randomly. Each search tree is used to generate only one solution. To generate n solutions, we generate n random search trees. The solutions from this approach are random in the sense that every solution has a non-zero probability of being found. However, the probability distribution is not uniform.

5.3 Generation of Random Models

Model generation by searching random solutions to the constraints introduces some important challenges such as the potential production of invalid models. A model is considered invalid if it does not meet the language requirements. In fact, even if the generated models satisfy the syntactic constraints of the modeling language, they might still not satisfy the semantic logic behind.

We may fail to encode the entire semantics of the modeling language in Clafer. We are limited to the description of the properties of the modeling language in UML meta-models.

In our study, SCADE is a synchronous language that guarantees that the data flow is immediately computed at each cycle with no physical latency. A loop in the generated model violates this rule, and the checker tool generates a causality error [5].

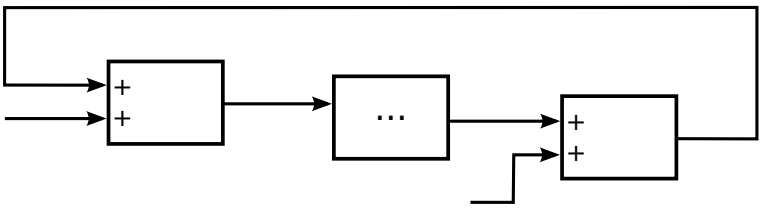

Figure 5.6: Example model with causality error

Such an error is not restricted to the use of SCADE and can occur in other synchronous modeling languages. To fix this issue, we developed a script that adds a delay block every time a cycle is detected in the generated model. The script runs in a loop the SCADE checker on the generated microbenchmarks, parses the errors and adds a delay block to break the cycle [9]. This technique aims to add a minimal number of delay blocks to break the cycle in the model. Few models had recursive loops due to the solver randomness. The scripts failed to resolve this issue by adding delays and the models were eliminated.

SCADE imposes some restriction such as maintaining the same data flow and prohibiting mixed data type. Such requirements were encoded as a set of constraints in Clafer.

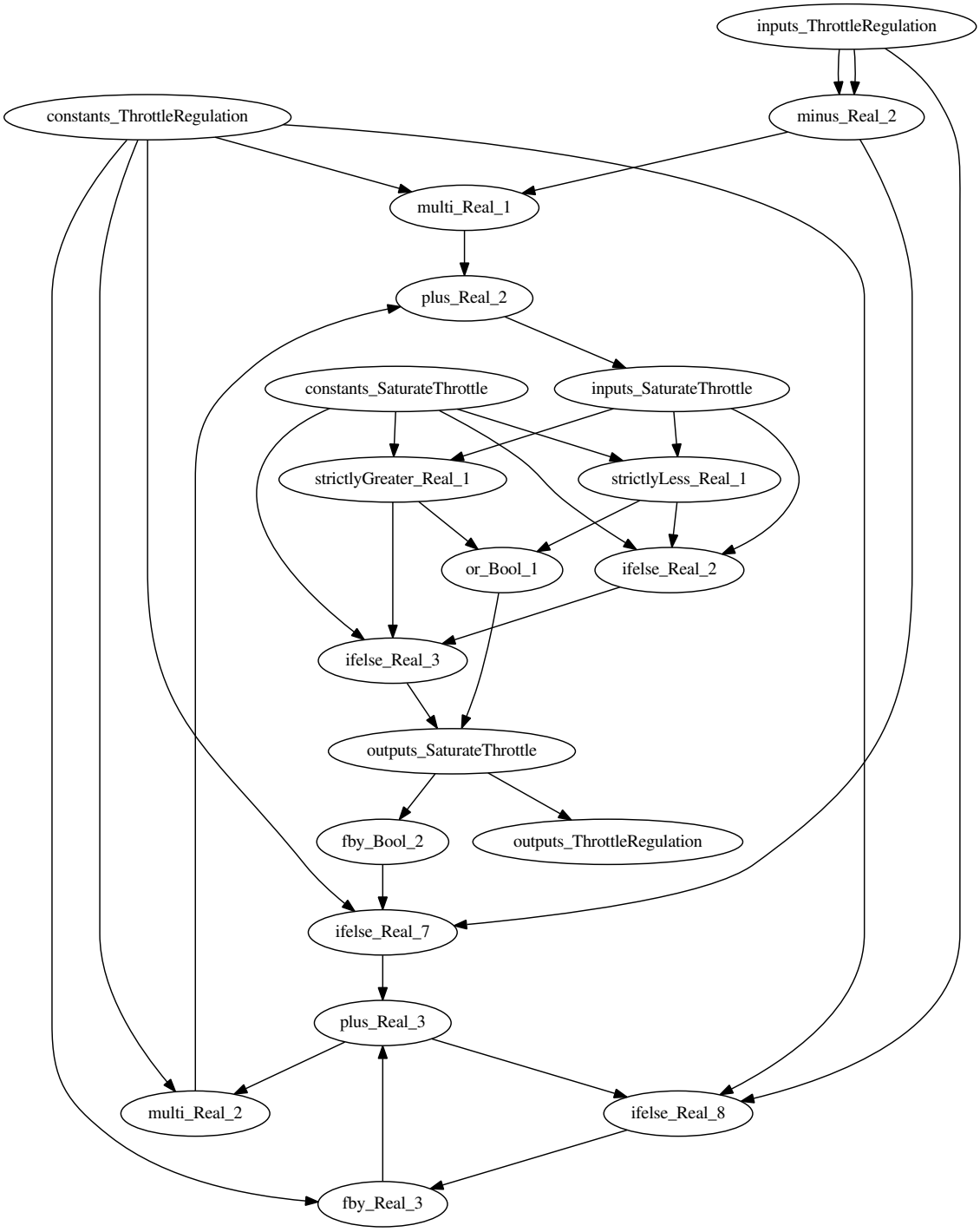

Figure 5.7: Throttle Regulation graph representation

Chapter 6

Experimentation

Given that we want to generate small-sized microbenchmarks that are easy to migrate, we fixed the size of the microbenchmarks. In that sense, the use of a constraint solver is not an issue.

6.1 Experimental Setup

The experiment was designed to run on DataMill. Several architectures are used for the benchmarking such as x86_64, i686, and ARM. We developed scripts to setup the experiment, run it and collect the data. The setup script uses GCC to compile the generated C code with no optimization flags. While running the experiment, we measured the execution-time of the microbenchmarks. The main function for the generated C code calls the main node of the microbenchmark. In other terms, the microbenchmark function would correspond to a call to one clock tick on the SCADE reduced model, and the data flow would pass through all the blocks of the model.

Figure 6.1 shows the procedure of measuring the execution-time. In the first step, we warmed up the system to compensate for measurement errors due to memory caches, buffers, etc. This step is important as it warms up the caches in the system. In the next step, we measured the execution of the microbenchmark code several times. The last step is the calculation of the measured execution-time, and it is based on the different measurements of the previous step. The measured data is stored in log files. Finally, the DataMill scripts compress the log files and collect the results of the different models submitted for benchmarking. After collecting the data from DataMill, we developed scripts

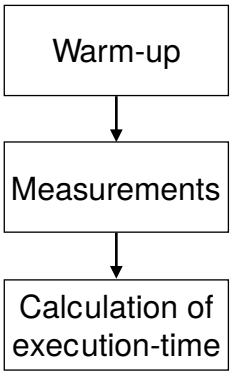

Figure 6.1: Execution-time measurement

to sort the results by metric, compute the mean execution-time of the microbenchmarks, and perform sanity checks on the data. For example, we checked for non-negative execution-time values. We also inspected the distribution of the measured data by producing q-q plots and histograms. Figure 6.2 is an example that shows the q-q plot of the mean execution-times of 1000 microbenchmarks of the Water Level application. The x-axis shows the theoretical mean values, while the y-axis shows the sample values. The linearity of the points in the plot suggests that the data is normally distributed which is reasonable to expect.

As the input values of the model can affect the execution-time, the benchmarking process was designed to average out this effect. Random input values are generated for each execution of each microbenchmark model. Each microbenchmark is executed many times (10000 times) with randomly chosen inputs. The execution-time of one microbenchmark is given by the mean value of these executions. The use of random inputs allows the exploration of different paths in the code to obtain an "average behavior" that properly describes the average execution and average code coverage. If a probabilistic model of the inputs for the given application or application domain is available, such model should be used as the source of random values for the inputs in the benchmarking process. This has the advantage that the execution is now statistically representative of the execution that the system will exhibit when in actual operation; thus, the prediction of the performance is specific to the real operating conditions under which the system will execute.

The hardware environment can also introduce bias in the execution performance. The execution-time can be mistakenly measured if executed on an exclusive architecture. The DataMill project solves this issue and proposes various architectures to benchmark the generated C code easily [12]. A set of scripts is used to setup the environment, run and

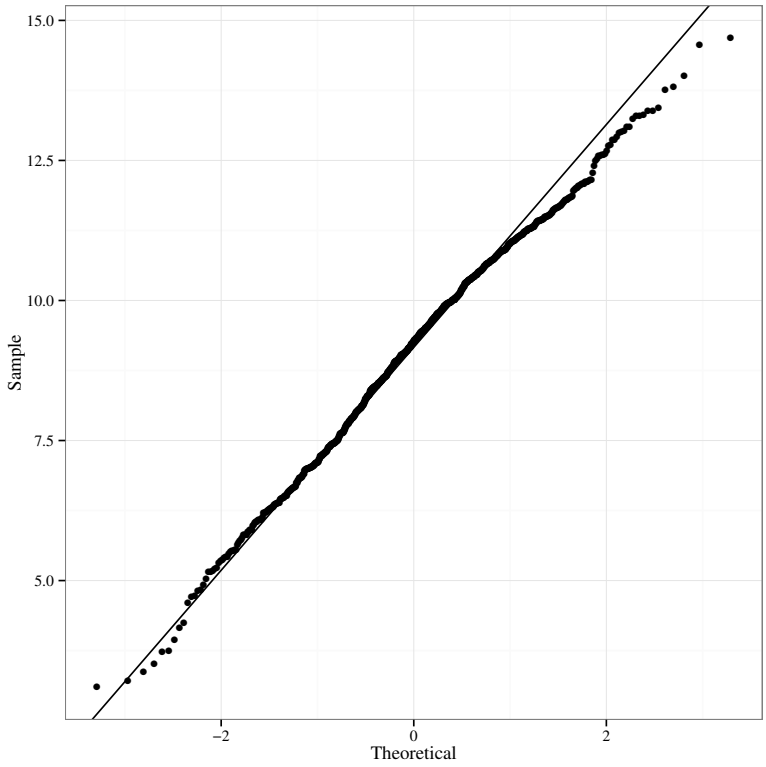

Figure 6.2: Q-Q plot of the mean execution-time of the microbenchmarks of the Water Level application

collect the execution-time across several architectures.

6.2 About the Experiments

We present in this work a framework to predict the performance parameters of model-based applications. The steps of the framework are presented in Section 2.2. We developed several scripts in this work to automate the process of the framework. The various scripts are available under the following bitbucket repository [8].

First, we developed TCL scripts to extract the structure and performance properties of the application models. We also developed scripts to generate the graph representation of the SCADE models as described in Section 5.1. After encoding the modeling language (SCADE) in Clafer, we developed scripts to compile the encoded meta-models, run the

constraint solver and generate random solutions. The latter are described as Clafer models. The effort of generating the microbenchmarks is equivalent to the effort of solving the constraints introduced by the extracted metrics. The size of the microbenchmarks is then correlated to the number of desired instances to generate. As the number of instances and constraints drives the inputs of the constraint solver, the microbenchmarks generation effort depends on the performance of the constraint solver. The generation of 2000 solutions for the static metric took around 10 minutes in our case on an Intel i3 processor machine with 6GB RAM.

We developed a script that parses Clafer generated models and generates textual models in SCADE 5 and SCADE 6. As the script ports the Clafer models into SCADE models, there is no human effort required in the migration process. The automation of this process allows us to have straightforward and inexpensive migration. We also developed scripts to fix causality issues as described in Section 5.3. After fixing the causality errors and ensuring that the models are valid, a set of scripts is developed to run KCG and generate the C code from the microbenchmarks. Another script is used to parse the inputs of the random microbenchmarks and generate the C code for the function that feeds the inputs with random inputs.

During the benchmarking process, we run, for each metric, 1000 generated fingerprints 1000 times with random inputs. We detailed the process in Section 6.1. A DataMill experiment benchmarks all the microbenchmarks generated using the several metrics and the validation application under both versions of SCADE. We execute the developed scripts on DataMill to setup the benchmarking experiment, run it and collect the data. A DataMill experiment takes around two days to finish the benchmarking process and report the collected data. We developed scripts to perform sanity checks and analyze the collected data.

6.3 Estimating Performance Parameters

Let $\mathcal{P}$ be a model with $|\mathcal{P}|$ blocks, and consider toolchains $\mathcal{T}_1$ and $\mathcal{T}_2$. Let $b_1, b_2, \cdots, b_c$ denote the C classes of blocks in the toolchains. For example, in SCADE, they would represent addition blocks, multiplication blocks, logical AND blocks, etc. Let $\tau_k^{(v)}$ be the execution-time of the fragment of code generated for blocks of type b_k under toolchain $\mathcal{T}_v$. We should expect execution-time for a particular type of block to be different under each toolchain, since the code generator is different. On the other hand, we assume a fixed execution-time for each type of block under the same toolchain, regardless of configuration and interaction with neighboring blocks. This is a reasonable approximation if the code

generator is not highly optimizing, which is the case for tools that generate safety-critical qualified code such as SCADE.

Let p_k denote the fraction of blocks of type b_k in $\mathcal{P}$, and let $T^{(v)}$, with $v = 1, 2$ be the average execution-times of the model's main function generated by each of the toolchains $\mathcal{T}_v$. The averages are considered over the population of all possible models with $|\mathcal{P}|$ blocks that maintain the fractions p_k for each type of block. Notice that data-flow based models such as SCADE have a directed acyclic graph (DAG) representation [11]. Evaluation of the model's main function requires traversal of the graph, either in a breadth-first traversal until reaching all of the outputs or as a topological sort. In either case, the complexity of the operation is $\mathrm{O}(V + E)$ where V is the number of vertices, and E is the number of edges [11]. A further observation is that since blocks have inputs and outputs, and outputs cannot be "short-circuited" together, then each input can only be connected to one output. Since each block has $\mathrm{O}(1)$ inputs, then $E = \mathrm{O}(V)$, and thus evaluation of $\mathcal{P}$'s main function takes $\mathrm{O}(|\mathcal{P}|)$ operations. Thus, we have

$$T^{(1)} = |\mathcal{P}| \sum_{k=1}^{C} \alpha_k^{(1)} p_k \tau_k^{(1)} \tag{6.1}$$

$$T^{(2)} = |\mathcal{P}| \sum_{k=1}^{C} \alpha_k^{(2)} p_k \tau_k^{(2)} \tag{6.2}$$

where the values $\alpha_k^{(v)}$ account for the average fraction of activity that each type of block in the model is exercised (including the multiplicative constant hidden in the big-Oh notation). If we consider a probability distribution of the inputs that does not vary for the different models, then clearly the values $\alpha_k^{(v)}$ are fixed and determined by the various compatible ways to connect outputs of blocks to inputs of other blocks and the average paths of propagation of input data (which depend both on the structure of the model and the data).

Consider models with $|\mathcal{P}'|$ blocks that maintain the fractions of each type of block, $p_1, p_2, \cdots, p_c$. It is reasonable to expect that the values of each $\alpha_k^{(v)}$ will be the same for models with $|\mathcal{P}'|$ blocks, since the fractions of each type of block p_k affect the possible configurations in which the sets of blocks can occur. Then, the average execution-times

$T'^{(v)}$ for models of size $|\mathcal{P}'|$ are

$$T'^{(1)} = |\mathcal{P}'| \sum_{k=1}^{C} \alpha_k^{(1)} p_k \tau_k^{(1)} \tag{6.3}$$

$$T'^{(2)} = |\mathcal{P}'| \sum_{k=1}^{C} \alpha_k^{(2)} p_k \tau_k^{(2)} \tag{6.4}$$

From equations (6.1), (6.2), (6.3) and (6.3), we obtain

$$\frac{T^{(2)}}{T^{(1)}} = \frac{T'^{(2)}}{T'^{(1)}} = \frac{\sum_{k=1}^{C} \alpha_k^{(2)} p_k \tau_k^{(2)}}{\sum_{k=1}^{C} \alpha_k^{(1)} p_k \tau_k^{(1)}}$$

$$\Rightarrow \quad T^{(2)} = T^{(1)} \frac{T'^{(2)}}{T'^{(1)}} \tag{6.5}$$

If $\hat{T}^{(1)}$ and $\hat{T}^{(2)}$ are the execution-times for a given model (as opposed to the average execution-time over all possible models of this size), then Equation (6.5) yields an approximation, allowing us to obtain an estimate of the execution-time under $\mathcal{T}_2$ given a statistical representation of $T'^{(1)}$ and $T'^{(2)}$:

$$\hat{T}^{(2)} \approx \hat{T}^{(1)} \frac{T'^{(2)}}{T'^{(1)}} \tag{6.6}$$

Estimation of the variance is done in a similar way; the variances of $T^{(v)}$ are the result of the variances of the individual execution-times for the blocks, $\tau_k^{(v)}$, since all of the other terms are constants, provided that the fractions of the types of blocks are preserved. Thus, assuming that the variables $\tau_k^{(v)}$ are uncorrelated, we have:

$$\mathrm{Var}\left(T^{(v)}\right) = |\mathcal{P}| \sum_{k=1}^{C} \beta_k^{(v)} \mathrm{Var}\left(\tau_k^{(v)}\right) \tag{6.7}$$

where the values $\beta_k^{(v)}$ depend exclusively on the values of $\alpha_k^{(v)}$ and the probability distribution of the $\tau_k^{(v)}$ variables.

Following a reasoning identical to that for the execution-time, we obtain a similar formula for the estimation of the variance of the execution-time under the new toolchain:

$$\mathrm{Var}\left(\hat{T}^{(2)}\right) \approx \mathrm{Var}\left(\hat{T}^{(1)}\right) \frac{\mathrm{Var}\left(T'^{(2)}\right)}{\mathrm{Var}\left(T'^{(1)}\right)} \tag{6.8}$$

Estimating Ratios from $\mathcal{T}_1$ to $\mathcal{T}_2$

Equations (6.6) and (6.8) give us an estimator for the parameters for models of one size based on the ratio of the parameters for models of a different size when migrated from $\mathcal{T}_1$ to $\mathcal{T}_2$. Thus, we estimate these ratios based on sampling them through multiple randomly generated fingerprint models of a fixed size.

Each randomly generated model is executed multiple times with randomly selected input data to obtain an estimate of the execution-time's mean and variance for the particular fingerprint model. We repeat this under both toolchains, to obtain one sample of the ratio between the parameters (mean and variance) under both toolchains. Assuming that both mean and variance, seen as random variables with respect to the population of all fingerprint models, follow a Gaussian distribution with non-zero mean, we empirically verified that their ratio follows a Gamma distribution. Thus, the mean of the samples obtained for each fingerprint model provides an adequate estimator for the required parameters.

Chapter 7

Results

We present and discuss in this chapter the results of our case study benchmarks. We tried our approach on several applications provided as part of the SCADE software. We refer to these applications as validation models as they are provided as example applications, and the models are available in both versions of SCADE. We used the SCADE example applications Cruise Control and Water Level. We developed another validation application that used a limited set of blocks; it used only Math operators such as addition and multiplication. For these applications, we had the models under SCADE 5 and the migrated models under SCADE 6.

We first discuss the results of the static ratio metric for the different validation applications, we then focus on the comparison of the various metrics presented in Section 5.1 for the same validation application Cruise Control.

The microbenchmarks ratios represent the estimated results using our approach following the inexpensive path of Figure 2.1. The application ratios represent the ratios of the migrated models by following the expensive path of Figure 2.1. We calculate the mean execution-time of the different executions with random inputs. The application ratio is the mean execution-time of the entire application in SCADE 6 divided by the mean execution-time of the migrated application in SCADE 5. We compute the ratio of the execution-time for each microbenchmark. We refer to the mean value of the ratio of the microbenchmarks as the microbenchmarks ratio. We used the R boot package to compute the ratios and the 95% confidence intervals of the microbenchmarks [7].

Table 7.1 shows the mean execution-time benchmarking results of the validation models Math operators application, the Water Level application, and the Cruise Control application. The rows show the target architectures that we used in our benchmarks, and the

columns show the application ratio and the microbenchmarks ratio. The ± refers to the margin of error computed by 95% confidence interval. These results are obtained by several executions of the C code with random inputs after a warm-up phase of the system as explained in Chapter 6.

The Math operators application has the smallest difference between the estimated ratio (the microbenchmarks ratio) and the actual ratio value (the application ratio) in both architectures. This suggests that complex models may be subject to lower accuracy in the prediction. For example, the introduction of multiple execution paths by conditional blocks could justify such a difference. The benchmarking results suggest that the generated models were executed faster in SCADE 6 than SCADE 5. The entire application shows the same aspect for the evaluation of the execution-time. The above results show that the estimated ratios are consistent with the actual application ratios. Both the estimated and the migrated results show a speedup of the models when migrated to SCADE 6.

	Architecture	
	i686	x86_64
Math Operators		
Application Ratio	0.724 ± 0.0007	0.664 ± 0.002
Microbenchmarks Ratio	0.789 ± 0.146	0.592 ± 0.012
Water Level		
Application Ratio	0.489 ± 0.001	0.454 ± 0.001
Microbenchmarks Ratio	0.421 ± 0.008	0.240 ± 0.008
Cruise Control		
Application Ratio	0.791 ± 0.011	0.325 ± 0.001
Microbenchmarks Ratio	0.444 ± 0.011	0.292 ± 0.013

Table 7.1: Execution-time ratio results

Even though the accuracy of the estimates is not too high, we notice that the framework gives a reasonable prediction of the performance evolution of the application. We notice that the 95% confidence intervals are tight which suggests that the effect of measurement errors and noise does not play a significant role in our results.

We can estimate that our method prediction is off by approximately 28% with respect to the correct performance for the migrated application (the geometric mean of actual ratio to predicted ratio across the validation applications is 1.28). A prediction accuracy of 28% can be useful in practice, since we can get severe performance regressions when migrating a model from one tool to another. Indeed, we observed performance variations between

toolchains of over 400%; for example, for the microbenchmarks for the Water Level model, the ratio for the same microbenchmark executed under SCADE 5 vs. executed under SCADE 6 varied from approx. 0.2 to 0.9. Predicting performance change within 28% accuracy is a good starting point in the decision-making process. Furthermore, for all of the models that we used in our experiments, our technique correctly predicted whether performance would improve or deteriorate; we claim that trend prediction is as important or even more so than the exact percentage.

We show the results of the variance prediction in Table 7.2. The application variance ratio is the ratio obtained for the validation applications available under the two versions of SCADE. The microbenchmarks variance ratio is the ratio of the variance of the entire microbenchmarks. The microbenchmarks used in this experiment are the same that were used for the execution-time prediction of Table 7.1. The figures that we obtained by extracting the variance results from the experiments were unexpected and looked unreasonable. One potential reason for this is the completely different way in which SCADE 5 and SCADE 6 handle the conditional blocks; from our observations of samples of generated code in both versions, conditional blocks in SCADE 5 involve execution of both branches followed by evaluation of the condition to choose one of the two already computed results. In SCADE 6, conditionals lead to optimized code: the condition is evaluated first, and only one of the two branches of the if-else is executed. This could have a profound impact on the variance of the execution-time, and, in particular, could have an effect on the accuracy of the prediction algorithm.

	Architecture	
	x86_64	i686
Math Operators		
Application Variance Ratio	0.3048	0.3964
Microbenchmarks Variance Ratio	0.0495	4.4482
Water Level		
Application Variance Ratio	0.2558	0.0245
Microbenchmarks Variance Ratio	0.1042	19.2726
Cruise Control		
Application Variance Ratio	0.132	0.5426
Microbenchmarks Variance Ratio	0.1783	0.2626

Table 7.2: Variance results

We focus on the comparison of the ratio and network metrics. Table 7.3 shows the

collected results for the different metrics. These results were collected through the same benchmarking process as detailed in Chapter 6.

	Architecture		
	x86_64	i686	ARM
Application Ratio	0.2515 ± 0.0011	0.7822 ± 0.0125	0.206 ± 0.0059
Static ratio of C code			
Microbenchmarks Ratio	0.4922 ± 0.0014	1.2045 ± 0.0789	0.6914 ± 0.0056
Static ratio of SCADE model			
Microbenchmarks Ratio	0.2822 ± 0.0092	0.4446 ± 0.0093	0.467 ± 0.0049
Dynamic ratio			
Microbenchmarks Ratio	0.2677 ± 0.0014	0.3 ± 0.0024	0.5295 ± 0.0092
PageRank			
Microbenchmarks Ratio	0.415 ± 0.0025	1.1339 ± 0.1377	0.5977 ± 0.0065
Fan-out			
Microbenchmarks Ratio	0.4958 ± 0.0061	0.5047 ± 0.005	0.9386 ± 0.5538
Eigenvector			
Microbenchmarks Ratio	0.4822 ± 0.0034	0.3332 ± 0.002	0.4862 ± 0.0044
Closeness			
Microbenchmarks Ratio	0.4763 ± 0.0026	0.6239 ± 0.0395	0.6201 ± 0.0032
Betweenness			
Microbenchmarks Ratio	0.6354 ± 0.0049	0.4512 ± 0.0031	0.6164 ± 0.0057

Table 7.3: Benchmarking results for the Cruise Control application

We focus here on the trend prediction of the performance parameters with our approach. We show in Table 7.3 the results of only one validation application Cruise Control. However, the results would be similar for the different validation applications as shown with the static ratio metric in Table 7.1. We benchmarked the Cruise Control application and the microbenchmarks generated with the different metrics on DataMill infrastructure. We used x86_64, i686 and ARM as target architectures.

Based on the results of Table 7.3, we can claim that the execution of the SCADE 6 applications is faster than the SCADE 5 ones. The results of Table 7.3 support the results collected in Table 7.1. All the metrics shows the same trend; there is a speedup in the execution of SCADE 6 models comparing to the SCADE 5 ones.

We present in Table 7.4 the geometric means of the actual ratio to the prediction ratio of

the different metrics. The equation is presented in (7.1), where x_i is the microbenchmarks ratio of the metric as presented in Table 7.3 and y_i is the application ratio.

$$\frac{\left(\prod_{i=1}^{n} x_i\right)^{(1/n)}}{\left(\prod_{i=1}^{n} y_i\right)^{(1/n)}} \tag{7.1}$$

We qualify the metrics with results closer to 1 as the best metrics to perform the estimates, because the geometric mean of the benchmarks of the microbenchmarks is closer to the geometric mean of the benchmarks of the validation application.

Metrics	Geometric Mean
Static ratio of C code	2.16
Static ratio of SCADE model	1.13
Dynamic ratio	1.02
PageRank	1.90
Fan-out	1.79
Eigenvector	1.24
Closeness	1.65
Betweenness	1.63

Table 7.4: Geometric means of the estimates with different metrics

The network metrics performed only moderately well with the SCADE modeling language. The ratio estimates are around the same interval [0.5,1]. However, we notice that the static ratio extracted at the model level, the dynamic ratio, and eigenvector metrics performed well comparing to the others. We expect the metrics to perform differently with other modeling languages. In fact, the estimates collected for each metric would depend on the modeling language and the target architecture. Given the application domain, the user would choose and tune the metrics to define the most efficient one (the most representative of the application models), implement it and use it in the prediction process.

The dynamic ratio metric seems to be the best metric in our case. The predicted estimates with the x86_64 and ARM architectures are the closest to the benchmarks of the migrated Cruise Control application. We believe that, in general, dynamic metrics would

perform better as they capture the characteristics of the application at run time. The execution-time would depend on the execution paths of the application. The extraction and benchmarking of the dynamic metric would require realistic input data. We will discuss this point in details in Chapter 8.

Comparing the two methods of the extraction of the static ratio metric as explained in Subsection 5.1.1, our benchmarks show that the estimates of the static ratio extracted at the model level are closer to the migrated application results. We suspect that the KCG code generator invokes some optimization. There are many hidden factors related to this particular difference between the C code and the model. An extension of this work might focus on this topic.

Chapter 8

Discussion

In this chapter, we discuss some of the important aspects that we believe have a critical impact on the validity and usability of our approach. Some of these issues could indeed represent threats to this validity. They could be related to the modeling language and the development toolchains. Below are some of the issues that we have considered:

- **Validity of software metrics**. The software metrics used to extract the data about the application models might not truly capture the application structure and architecture. For example, we might fail to detect the blocks patterns presented in the application. These patterns may affect the execution-time performance. Indeed, the interaction between connected blocks could have an important impact on the generated code's computational efficiency given low-level aspects such as cache, pipelining, or other hardware-related aspects. This makes us believe that there may be hard-to-capture underlying patterns in the structure that could have a significant impact on the overall performance. Thus, the accuracy of the prediction might benefit if the used metric would capture those patterns. The ratio and network metrics proved to give reasonable results in our case studies, but we have to acknowledge the possibility that it might be insufficient for some other cases. We are convinced that this is the most important area that requires future work. Notice, however, that our proposed framework is extensible, and practitioners can use any other metrics that they have developed, and that may produce good results in the contexts being used.
- **Dynamic ratio metric**. The dynamic metric showed a slight improvement in the prediction of the performance parameters using the ratio metric. We believe that the dynamic metric would perform better if we used realistic input data collected from the

domain application instead of randomly generated inputs. Random value generation functions are used at the C code level to feed the models with random inputs. The inputs do not follow a particular distribution and are randomly generated. The use of random inputs triggers most of the model paths during the benchmarking, which provides an accurate average of the execution-time. Knowing the environment domain of the application, we could simulate the distribution of the inputs which narrows the simulation errors and improve our prediction results. The distribution of the inputs would trigger the models differently, as in the real environment, and we would predict the execution-time more precisely. We emphasize that the generated microbenchmarks based on dynamic metrics would require realistic data to predict the execution-time of the application.

- **Validity under different architectures**. We only benchmarked our experiments on a limited number of architectures. There are other architectures like MIPS, PowerPC, etc. However, ARM, Intel x86_64, and i686 cover an important fraction of the target audience for our method. Additional architectures will be used in future work.

- **Non-uniform random solutions**. The constraint solver does not generate perfectly random solutions, which could affect the accuracy of the prediction. We inspected the generated models and found a reasonable diversity in the models; we trust that this was not an issue in our experimentation. Moreover, generation of uniform random solutions by a constraint solver is a known complex problem, so our framework could certainly benefit from any progress that the AI community may make in this area.

- **Variance prediction sensitive to modeling tools**. As already mentioned in Chapter 7, the modeling tool could handle conditional and similar blocks in very different ways that could affect the prediction of the variance. This is a potentially critical aspect that we believe requires future work: changes in the variance may have an effect on estimates of WCET analysis if they were performed using measurement-based approaches. Even though this WCET analysis has to be done on the migrated model, the ability to accurately predict changes in the variance provides important information with respect to the risks that the migration could involve.

- **Restricted size of microbenchmarks**. Conceivably, the sizes that we chose for the microbenchmarks — which obey restrictions in the capacity of the constraint solver — could limit the accuracy of the predictions. The intuition is that larger microbenchmark models could have better ability to capture more complex characteristics. As constraint solvers become increasingly powerful, our framework could

in turn benefit from any such advances.

8.1 Contributions

The main contribution of our work is the established framework that can be used to predict performance parameters without porting the entire application from one toolchain to another. We present the steps that users can follow to implement our framework in Section 2.2.

We also wrote scripts to implement the framework and predict the performance parameters of an application under SCADE 6 given the SCADE 5 models. We developed and automated the experimentation workflow for validation of the framework. We developed scripts to extract various metrics that describes the characteristics of the application models. First, the scripts extract several properties of the expressions and equations of the SCADE models. Second, we generate random microbenchmarks and migrate them from one toolchain to another. Last, we run benchmarking experiments and analyze the data to extrapolate an estimate of the execution-time of the application under SCADE 6. A detailed description of the developed scripts is provided in Section 6.2.

We provide in this work an initial implementation of the framework of our approach. Our predictions for the SCADE systems, presented in Chapter 7, are the initial results of our framework. The developed scripts and the tools for the SCADE language are the first concrete implementation of our framework and are subject to improvement with future work. We believe that our framework would benefit from future work focusing on the characterization and the extraction of the performance relevant properties as software metrics.

8.2 Generalization

The goal of this section is to discuss the generalization claim of the use of our approach with different modeling language. In this work, we focused on the SCADE modeling language. However, we would like to emphasize that our approach can be used for any modeling language such as LabView, Simulink, SysML, ModelSim, etc. We show for example the process of generating the microbenchmarks for estimation purposes with Simulink modeling language.

We believe that our tool can be generalized and used with any modeling language. In fact, the benchmark generator uses a model similar to UML class models and Object Constraint Language (OCL) and thus can be used to model the complete syntax of any language. The grammar of any modeling language can be captured by our tool to generate syntactically valid models. We show in this section the application of our approach with the Simulink modeling language. As explained in Section 5.3, the random generation may require the use of checking procedures to confirm the validity of the generated microbenchmarks. Our approach benefits from the nature of synchronous languages (such as limited use of iteration); the applicability of our approach to enterprise systems may not be obvious.

Section 2.2 presents the steps suggested by our approach; we will detail in this section its application on a Simulink model. Figure 8.1 shows an example of a Simulink model. We can develop a set of scripts to extract several metrics. We showed in Section 5.1 few metrics that can be used to identify the characteristics of the models structure and capture the main properties affecting the execution-time.

Table 8.1 shows the occurrence and the static ratio of each type of blocks used in the model of Figure 8.1. To extract the network metrics, we generate the graph representation. Figure 8.2 is the equivalent graph of Figure 8.1. Table 8.2 shows the extracted results of the network metrics for the Simulink example model. After extracting the metrics values, we encode them as constraints in the Clafer language as presented in Listing 8.1.

```
1 Substract : MathBlock 2
2 Gain : GainBlock 3
3 Integrator: IntegratorBlock 2
```

Listing 8.1: Static ratio metric data encoded in Clafer

The constraints are the cardinality of the instances in this example. We emphasize that the user should define the classes MathBlock, GainBlock, IntegratorBlock to represent the characteristics of the Simulink language. The solutions of the constraint solver are the application-specific microbenchmarks.

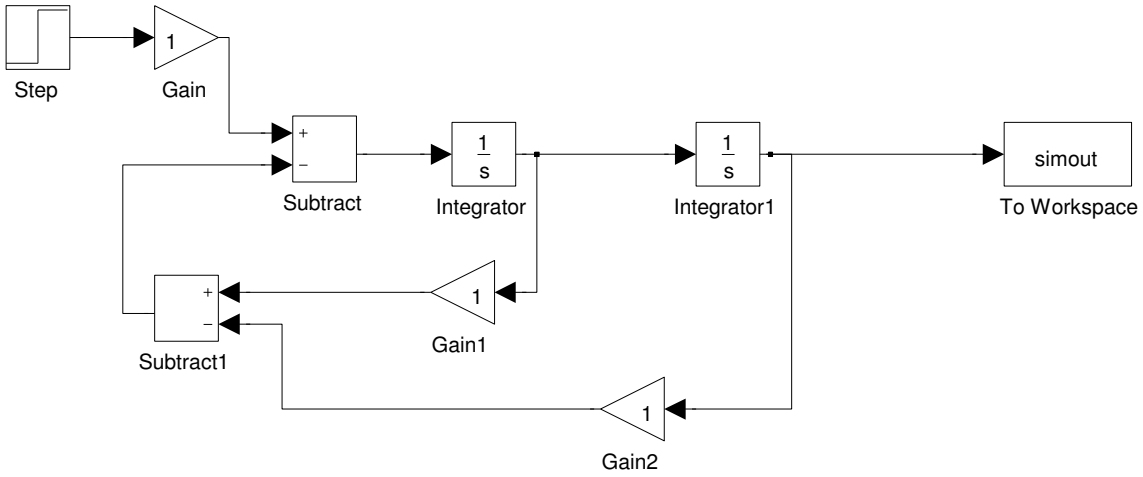

Figure 8.1: Example of Simulink model

	Occurrence	Ratio
Substract	2	28.58%
Gain	3	42.86%
Integrator	2	28.58%

Table 8.1: Extracted static ratio metric of Simulink example

	Fan-Out	PageRank	Closeness	Betweenness	Eigenvector
Integrator1	11.111	11.085	20.833	16.071	38.776
Integrator	22.222	21.751	34.722	33.929	46.294
Substract	11.111	14.308	24.038	30.357	27.206
Gain1	11.111	11.085	20.833	1.786	38.776
Substract1	11.111	23.424	26.042	37.500	55.268
Gain2	11.111	11.262	22.321	12.500	32.480
Gain	11.111	3.405	23.684	10.714	0
outputs_Simulink	0	1.840	0	0	0
inputs_Simulink	11.111	1.840	23.558	0	0

Table 8.2: Extracted network metrics of Simulink example

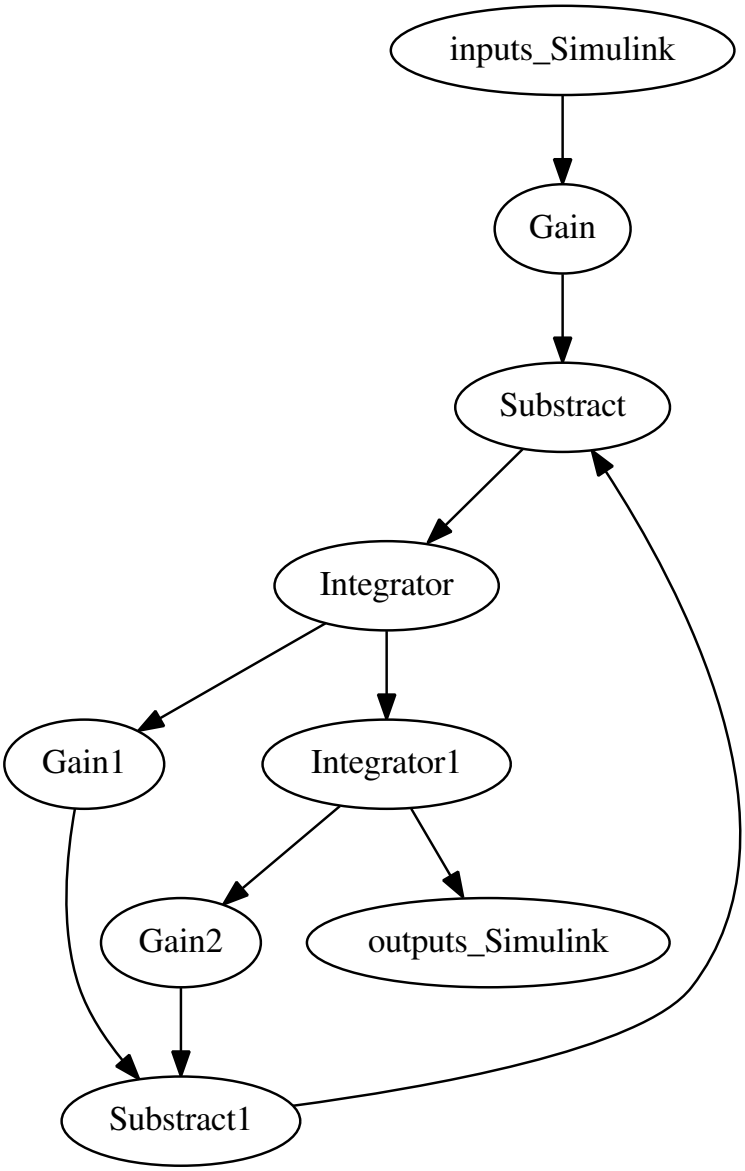

Figure 8.2: Graph representation of Simulink model

Chapter 9

Conclusions

We presented in this work a framework to predict the execution-time performance parameters of model-based applications under different toolchains. Our approach has low cost as we avoid the migration cost by automating the migration of the models between the two toolchains. We can predict the performance parameters with minimal porting efforts expressed as the number of changes in the model.

To follow our framework, the application should be analyzed to extract software metrics that are relevant to the modeling language. The metric should be encoded in a constraint solver to generate application-specific microbenchmarks. The generated microbenchmarks are representative of the application, and the benchmarking of the microbenchmarks provides an estimate of the execution-time of the application under the two toolchains.

To illustrate our framework, we presented a SCADE Systems case study. We verified that our approach produced performance predictions that are reasonably close to the performance that we measure with concrete results.

As a future research direction, we think that this work can be extended by studying the industrial practicality of our approach. We can then study the impact of the inputs on the models. This may be important as particular models in a given industrial application may have known distributions for the input data. A more detailed analysis of the effect of the distributions could have a positive impact in the applicability of our methodology. Furthermore, this could be useful when applying the technique to application domains instead of specific applications — for a given domain, typical distributions of input data may be known. Comparing and combining our approach with analytical approaches are important future research directions.

List of Acronyms

MDD Model Driven Development

IE Information Extraction

WCET Worst-Case Execution Time

SAT Boolean SAtisfiability

SMT Satisfiability Modulo Theories

CSP Constraint Satisfaction Problem

GML Graph Modeling Language

DOT Graph Description Language

TCL Tool Command Language

UML Unified Modeling Language

OCL Object Constraint Language

APPENDICES

```
1  // general blocks
2
3  abstract Block
4    isOutput-> int
5    [isOutput=0||isOutput=1]
6  abstract BlockWithIntOutput : Block
7  abstract BlockWithBoolOutput : Block
8  abstract BlockWithRealOutput : Block
9
10 // array blocks
11
12 abstract BlockWithArrayOutput : Block
13     arrayDimension -> integer
14     [arrayDimension > 0]
15
16 abstract BlockWithIntArrayOutput : BlockWithArrayOutput
17 abstract BlockWithBoolArrayOutput : BlockWithArrayOutput
18 abstract BlockWithRealArrayOutput : BlockWithArrayOutput
19
20 // matrix blocks
21
22 abstract MatrixBlock : Block
23     firstDimension -> integer
24     [firstDimension > 0]
25     secondDimension -> integer
26     [secondDimension > 0]
27
28 abstract BlockWithIntMatrixOutput : MatrixBlock
29 abstract BlockWithRealMatrixOutput : MatrixBlock
30 abstract BlockWithBoolMatrixOutput : MatrixBlock
31
32
33 //**********Math************ //
```

```
34
35 abstract MathBlockInt : BlockWithIntOutput  // block1.input may be connected
       to block2.input
36     i -> BlockWithIntOutput 0..2 // an input may be connected to an output
       of another block that has an BlockWithIntOutput  // up to 2 inputs for a
       block
37     [!(this in i.ref)] // looping is not authorized
38
39 abstract MathBlockReal : BlockWithRealOutput
40     i -> BlockWithRealOutput 0..2
41     [!(this in i.ref)]
42
43 //**********Comparison*********** //
44
45 abstract ComparisonBlockInt : BlockWithBoolOutput
46     i -> BlockWithIntOutput 0..2
47
48 abstract ComparisonBlockReal : BlockWithBoolOutput
49     i -> BlockWithRealOutput 0..2
50
51 //**********Boolean*********** //
52
53 abstract BooleanBlock : BlockWithBoolOutput
54     i-> BlockWithBoolOutput 0..2
55     [!(this in i.ref)]
56
57 abstract NOTBlock : BlockWithBoolOutput
58     i-> BlockWithBoolOutput 0..1
59     [!(this in i.ref)]
60
61 //**********Math Instances********** //
62 plus_Int : MathBlockInt *
63 plus_Real : MathBlockReal *
64 minus_Int : MathBlockInt *
65 minus_Real : MathBlockReal *
66 multi_Int : MathBlockInt *
67 multi_Real : MathBlockReal *
68
69 //**********Comparison Instances********** //
70 strictlyLess_Int : ComparisonBlockInt *
71 strictlyGreater_Real : ComparisonBlockReal *
72 greaterOrEqual_Int : ComparisonBlockInt *
73 different_Int : ComparisonBlockInt *
74 equal_Int : ComparisonBlockInt *
75 equal_Real : ComparisonBlockReal *
```

```
76
77 //**********Boolean Instances********** //
78 and_Bool : BooleanBlock *
79 or_Bool : BooleanBlock *
80 not_Bool : NOTBlock *
```

Listing 1: Minimal description of SCADE modeling language in Clafer syntax

```
1  digraph{
2  constants_SaturateThrottle->ifelse_Real_3;
3  ifelse_Real_3->outputs_SaturateThrottle;
4  inputs_SaturateThrottle->ifelse_Real_2;
5  constants_SaturateThrottle->ifelse_Real_2;
6  ifelse_Real_2->ifelse_Real_3;
7  or_Bool_1->outputs_SaturateThrottle;
8  inputs_SaturateThrottle->strictlyLess_Real_1;
9  constants_SaturateThrottle->strictlyLess_Real_1;
10 strictlyLess_Real_1->ifelse_Real_2;
11 strictlyLess_Real_1->or_Bool_1;
12 inputs_SaturateThrottle->strictlyGreater_Real_1;
13 constants_SaturateThrottle->strictlyGreater_Real_1;
14 strictlyGreater_Real_1->ifelse_Real_3;
15 strictlyGreater_Real_1->or_Bool_1;
16 }
```

Listing 2: DOT representation of Saturate Throttle model

References

[1] Aditya Agrawal, Gyula Simon, and Gabor Karsai. Semantic translation of simulink/stateflow models to hybrid automata using graph transformations. *Electronic Notes in Theoretical Computer Science*, 109:43–56, 2004.

[2] Andrew W. Appel. Verified Software Toolchain. In *Proceedings of the 20th European Conference on Programming Languages and Systems: Part of the Joint European Conferences on Theory and Practice of Software*, ESOP'11/ETAPS'11, pages 1–17, Berlin, Heidelberg, 2011. Springer-Verlag.

[3] Kacper Bak, Krzysztof Czarnecki, and Andrzej Wasowski. Feature and meta-models in clafer: mixed, specialized, and coupled. In *Software Language Engineering*, pages 102–122. Springer, 2011.

[4] S. Balsamo, A di Marco, P. Inverardi, and M. Simeoni. Model-based performance prediction in software development: a survey. *Software Engineering, IEEE Transactions on*, 30(5):295–310, May 2004.

[5] Grard Berry. SCADE: Synchronous Design and Validation of Embedded Control Software. In S. Ramesh and Prahladavaradan Sampath, editors, *Next Generation Design and Verification Methodologies for Distributed Embedded Control Systems*, pages 19–33. Springer Netherlands, 2007.

[6] Phillip Bonacich. Some unique properties of eigenvector centrality. *Social Networks*, 29(4):555–564, 2007.

[7] Angelo J Canty. Resampling methods in r: the boot package. *R News*, 2(3):2–7, 2002.

[8] Source code Bitbucket repository. https://goo.gl/5FwddW.

[9] Jean-Louis Colaço and Marc Pouzet. Type-based Initialization Analysis of a Synchronous Data-flow Language . *Electronic Notes in Theoretical Computer Science*, 65(5):65 – 78, 2002. SLAP'2002, Synchronous Languages, Applications, and Programming (Satellite Event of {ETAPS} 2002).

[10] S. D. Conte, H. E. Dunsmore, and Y. E. Shen. *Software Engineering Metrics and Models*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1986.

[11] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 2009.

[12] Augusto Born de Oliveira, Jean-Christophe Petkovich, Thomas Reidemeister, and Sebastian Fischmeister. Datamill: Rigorous performance evaluation made easy. In *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*, pages 137–148. ACM, 2013.

[13] Mrinal Kanti Debbarma, Swapan Debbarma, Nikhil Debbarma, Kunal Chakma, and Anupam Jamatia. A Review and Analysis of Software Complexity Metrics in Structural Testing. *International Journal of Computer and Communication Engineering*, 2:129–133, 2013.

[14] Florian Deissenboeck, Benjamin Hummel, Elmar Juergens, Michael Pfaehler, and Bernhard Schaetz. Model clone detection in practice. In *Proceedings of the 4th International Workshop on Software Clones*, pages 57–64. ACM, 2010.

[15] Francois-Xavier Dormoy. SCADE 6: a model based solution for safety critical software development. In *Proceedings of the 4th European Congress on Embedded Real Time Software (ERTS'08)*, pages 1–9, 2008.

[16] Lydie du Bousquet, Michel Delaunay, Huy-Vu Do, and Chantal Robach. Analysis of testability metrics for Lustre/SCADE programs. In *Advances in System Testing and Validation Lifecycle (VALID), 2010 Second International Conference on*, pages 26–31. IEEE, 2010.

[17] K. Falkner, V. Chiprianov, N.J.G. Falkner, C. Szabo, J. Hill, G. Puddy, D. Fraser, A Johnston, M. Rieckmann, and A Wallis. Model-Driven Performance Prediction of Distributed Real-Time Embedded Defense Systems. In *Engineering of Complex Computer Systems (ICECCS), 2013 18th International Conference on*, pages 155–158, July 2013.

[18] Norman E Fenton and Martin Neil. Software metrics: successes, failures and new directions. *Journal of Systems and Software*, 47(2 – 3):149 – 157, 1999.

[19] Norman E. Fenton and Martin Neil. Software Metrics: Roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, pages 357–370, New York, NY, USA, 2000. ACM.

[20] Robert France and Bernhard Rumpe. Model-driven development of complex software: A research roadmap. In *2007 Future of Software Engineering*, pages 37–54. IEEE Computer Society, 2007.

[21] Linton C Freeman. A set of measures of centrality based on betweenness. *Sociometry*, pages 35–41, 1977.

[22] Linton C Freeman. Centrality in social networks conceptual clarification. *Social networks*, 1(3):215–239, 1979.

[23] Mathias Fritzsche and Jendrik Johannes. Putting Performance Engineering into Model-Driven Engineering: Model-Driven Performance Engineering. In Holger Giese, editor, *Models in Software Engineering*, volume 5002 of *Lecture Notes in Computer Science*, pages 164–175. Springer Berlin Heidelberg, 2008.

[24] John Grundy, Yuhong Cai, and Anna Liu. Softarch/mte: Generating distributed system test-beds from high-level software architecture descriptions. *Automated Software Engineering*, 12(1):5–39, 2005.

[25] Mats PE Heimdahl. Safety and software intensive systems: Challenges old and new. In *2007 Future of Software Engineering*, pages 137–152. IEEE Computer Society, 2007.

[26] Sallie Henry and Dennis Kafura. Software structure metrics based on information flow. *Software Engineering, IEEE Transactions on*, (5):510–518, 1981.

[27] Reid Holmes and David Notkin. Identifying program, test, and environmental changes that affect behaviour. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 371–380. ACM, 2011.

[28] Yan Liu, Ian Gorton, and Alan Fekete. Design-level performance prediction of component-based applications. *Software Engineering, IEEE Transactions on*, 31(11):928–941, 2005.

[29] Guido Menkhaus and Brigitte Andrich. Metric suite directing the failure mode analysis of embedded software systems. In *ICEIS (3)*, pages 266–273, 2005.

[30] Tom Mens and Pieter Van Gorp. A Taxonomy of Model Transformation. *Electronic Notes in Theoretical Computer Science*, 152(0):125 – 142, 2006. Proceedings of the International Workshop on Graph and Model Transformation (GraMoT 2005) Graph and Model Transformation 2005.

[31] Kazuya Okamoto, Wei Chen, and Xiang-Yang Li. Ranking of closeness centrality for large-scale social networks. In *Frontiers in Algorithmics*, pages 186–195. Springer, 2008.

[32] K.A Ors and B. Laszlo. SCADE interpreter for measuring static and dynamic software metrics. In *Intelligent Systems and Informatics (SISY), 2013 IEEE 11th International Symposium on*, pages 123–128, Sept 2013.

[33] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: bringing order to the web. 1999.

[34] Gaurav Pandey, Jan Jelschen, and Andreas Winter. Towards quality models in software migration. *analysis*, 1:M2.

[35] Louis M Rose, Markus Herrmannsdoerfer, James R Williams, Dimitrios S Kolovos, Kelly Garcés, Richard F Paige, and Fiona AC Polack. A comparison of model migration tools. In *Model Driven Engineering Languages and Systems*, pages 61–75. Springer, 2010.

[36] Francesca Rossi, Peter Van Beek, and Toby Walsh. *Handbook of constraint programming*. Elsevier, 2006.

[37] M. Shepperd and D.C. Ince. A critique of three metrics. *Journal of Systems and Software*, 26(3):197 – 210, 1994.

[38] Adrian Smith, Arnaud Doucet, Nando de Freitas, and Neil Gordon. *Sequential Monte Carlo methods in practice*. Springer Science & Business Media, 2013.

[39] Ronald E. Walpole, Raymond H. Myers, Sharon L. Myers, and Keying Ye. *Probability & Statistics for Engineers & Scientists*. Prentice-Hall, Ninth edition, 2011.

[40] ANSYS SCADE website. http://www.ansys.com.

[41] Qi Zhu and Peng Deng. Design Synthesis and Optimization for Automotive Embedded Systems. In *Proceedings of the 2014 on International Symposium on Physical Design*, ISPD '14, pages 141–148, New York, NY, USA, 2014. ACM.