Model-driven Physical-Design Automation for FPGAs:  
Fast Prototyping and Legacy Reuse

Ciprian Teodorov and Loïc Lagadec *

Lab-STICC MOCS, CNRS UMR 6285, European University of Brittany, 20, avenue Le Gorgeu C.S. 93837 BP 809 29238 BREST Cedex 3, France

SUMMARY

The current integrated circuit technologies are approaching their physical limits in terms of scaling and power consumption, in this context, the Electronic Design Automation (EDA) industry is pushed towards solving ever more challenging problems in terms of performance, scalability and adaptability. Meeting these constraints needs innovation at both the algorithmic and the methodological level. Amongst academic EDA tools, Madeo toolkit has been targeting Field-Programmable Gate Array (FPGA) design-automation at the logic and the physical level since the late 90s. As many other long-living software, despite embedding valuable legacy, Madeo exhibits unwanted characteristics that penalize evolution and render the automation problems even more difficult.

This study presents a methodological approach to physical-design automation relying on model-driven engineering, which is illustrated through the incremental redesign of the Madeo framework. A benefit of this approach is the emergence of a common vocabulary to describe the EDA domain in a FPGA scope. A second advantage is the isolation of the optimization algorithms from the structural domain models. However, the main asset is the possibility to re-inject into the newly designed toolkit most of the legacy code.

The redesigned framework is compared to and scored against initial code-base, and demonstrates a regression-free remodeling of the environment with net improvements in terms of size and complexity metrics. As a consequence, the evolution capability is back on stage, and the domain-space exploration widens to the algorithmic axis. Copyright © 2013 John Wiley & Sons, Ltd.

KEY WORDS: model-driven; software-evolution; physical-design; FPGA

1. INTRODUCTION

Object-oriented design enabled, for years, the development of feature-rich software environments, which propose flexible and extensible solutions for complex problem spaces, such as electronic design automation (EDA). The Madeo framework [1, 2] is such an environment, automating the complete design flow of FPGA-based application development, from high-level application synthesis to the physical design of the circuit layout. As the result of 15 years of proactive research, the Madeo infrastructure cannot be described as a single software solution. Instead, it is a compilation of a number of different projects focusing on a variety of aspects in the FPGA CAD domain. These projects correspond directly to the research topics under investigation at different times during its existence.

The high-complexity of today’s reconfigurable architectures poses huge constraints in terms of automation. To tame this complexity, the FPGA application synthesis is decomposed in a number
of relatively independent problems, that are to be solved and the results integrated into the final solution. Most of these problems are NP-complete and typically solved heuristically. Moreover, the optimization objectives vary and the literature is very rich in highly optimized solutions for specific configurations.

To survive in such a demanding context, a software solution has to be very adaptable, flexible and it must support evolution over time. Exploiting the advantages of the Smalltalk language and environment, the Madeo infrastructure provides an unified and complete solution for FPGA application synthesis and related problems [3, 4]. From an evolution perspective, three principal evolution axes are present:

- Extending the domain model to integrate new domain-specific concepts;
- Adding missing functionality to improve the domain coverage and the automation level;
- Adding new functions, similar to the ones already present, to give the user a choice, especially in terms of the variety of heuristics and optimization objectives available.

The OO design methodology [5, 6] provides largely accepted mechanisms for extending the domain model, the inheritance and the composition. The design of the Madeo infrastructure, as a framework, exploits inheritance for easily adding new domain concepts while maintaining a high-degree of cohesion and code sharing. In terms of adding new behavior, the OO design methodology offers a variety of choices relying on overloading and polymorphism. However, it is not clear on the preferred way to functionally extend a system with behavior that does not align well with class hierarchy extensions, especially since the inheritance mechanism can be used for code sharing and for classification [7]. Moreover, in a research context, the underlining idea is to rapidly produce prototypes focused on specific research needs. In consequence, after years of evolution and improvement, our software environment got bloated with many different projects (at different maturity levels).

From a software engineering perspective [8] our principal goal is to preserve our legacy codebase, which gives us a strategic advantage over other research groups. This legacy code-base drives the innovation in our group by providing high-end solutions for many practical EDA problems. This enables us to focus on novel and challenging aspects of the integrated circuit design and automation. But at the same time, we need a methodological solution addressing the functional evolution aspects by isolating the core functionality and exposing clear interfaces. Moreover, such a methodological solution will open "the toolbox" making it accessible to a larger user base.

During the last years, model-driven engineering (MDE) and component-based development have become some of the most promising methodological approaches for tackling the software complexity and evolution problems. Based on the success stories of the MDE community, we focused our attention on finding an evolution methodology fitting our constraints, especially in terms of legacy-code reuse. Even though UML\(^3\) became the de-facto standard for model-based methodologies, in the case of our software projects, largely based on Smalltalk philosophy and tools, we believe that the Smalltalk community is still providing the right answers.

This study presents methodological and practical results for tackling the embarrassingly complex problems of integrated circuit (IC) physical design automation using software engineering techniques. These result are easily transposable to other application domains which exhibit similar characteristics, such as large number of domain concepts, the need for highly sophisticated algorithmic solutions, and adaptability to changing requirements. To the best of our knowledge, except our preliminary study in [9, 10], this proposition is the first MDE-based approach addressing the physical design automation problem. The principal contributions of this study are:

- The introduction of a novel conceptual framework, based on the MDE methodology, for the physical design aspects of the EDA flow. This framework stresses the need of decoupled software artifacts, and isolated concern-oriented software modules. This methodology doesn’t impose any constraint on the implementation language nor on the formalism of the core

---

\(^3\)Unified modeling Language
algorithms and heuristics needed, which is very important given the computational- and memory-intensive solutions needed in our context.

- The definition of an abstract vocabulary for structurally describing the applicative and architectural artifacts involved in the flow. The use of this abstraction level enables the creation of a common API to be used for implementing cross-cutting features, such as domain agnostic manipulations, querying, model serialization, etc.

- The presentation of a novel way for conceptually describing, characterizing and implementing physical design algorithms (e.g. placement, routing) using a transformation metaphor, based on which the algorithms are viewed as model-to-model composite transformations organized as hierarchical directed-acyclic graphs. This view helps the algorithm designer to focus on the specific problem to be solved while enabling fine-grained reuse of software.

- The analysis of the impact of modeling physical-design heuristics using the transformation metaphor in terms of development cost, performance and extensibility. The results show a reduction of the development cost for architecture-specific refinements. Besides, this approach enables fine grain performance tuning and offers a high flexibility for adding new functionality at the expense of higher memory consumption.

- The objective evaluation of the Madeo FPGA framework based on software engineering metrics. This evaluation gives an overview of the object-oriented Madeo system quantifying the complexity, the coupling and the usage of inheritance in this legacy toolkit. Our development policy was tuned based on the insight obtained by comparing the results with a statistically computed baseline of twenty other large Smalltalk projects.

- The presentation of an agile evolution strategy that was used to refactor the legacy code-base to fully functional intermediate solutions. The main purpose of this strategy is to incrementally improve the design towards the model-driven structure presented in this study without losing functionality at any step during this process.

This paper points out some interesting lessons. First, in spite of a continuous integration process, an in-depth redesign cannot be avoided but only delayed, as soon as the targeted domain is itself subject to major evolution. At this point, due to the programming approach that has been put into practice, the legacy should likely be highly valuable, and its loss unaffordable. There is no silver bullet, but we claim that the Smalltalk environment eases migration of such a mutable code; legacy can be split up into components to be further re-injected, one by one, into the new skeleton. By skeleton we refer to generated models, coming from a MDE compliant approach. This is made simple by the MDE framework that is itself written in Smalltalk, and which by using Fame [11] maintains a causal connection between meta and base levels. As a result, the new software is slimmer (modular/shorter), simpler (understandable/evolution capable) while showing performance improvements at the expense of higher memory footprint. Besides, it keeps alive the ability to embed legacy code on-demand. This ability is a key issue since it prevents functionality loss, and supports progressive rewriting.

However, this process must be quantified in terms of effort and expected benefits. This comes along with defining metrics. Meaningful metrics are hard to isolate for a broad audience, and even harder in the context of large domain-specific legacy code-base. However, instead of fixing intrinsic “best” metrics in an early stage, a clever option is to grab a set of metrics. Measuring the code refactoring and code redesign against these metrics offers a perspective view. As engineers in charge of such a redesign are experts, one can reasonably consider that the changes they operate over the code carry benefits (legacy skills). Observing these benefits gives back a multi-dimension signature. This signature leads to characterizing future evolutions, and favors an objective view of subjective feelings. As in any added-value centric activity, the human lays at the heart of the process, but our approach promotes using indicators as a tool to make the designer more confident and to enable him to operate faster.

The remainder of this study is organized as follows. Section 2 presents the state of the art in terms of MDE and circuit design automation. In Section 3, the Madeo FPGA toolkit is presented and evaluated using software engineering metrics. Section 4 starts by presenting the enabling technologies and experiences before showing the principal characteristics of our methodology.
Section 5 further describes our approach focusing on the domain-specific aspects of FPGA physical-design. Section 6 presents the design of a time-driven routing algorithm and uses it as a case study to show the principal characteristics of the transformation-based approach to algorithm design. Section 7 overviews some practical experiences with our agile approach to legacy refactoring, objectively quantifies the current state of the prototype, and compares the results with the evaluation of the Madeo legacy code-base. Section 8 reviews the core ideas presented in this study, before presenting some future research directions.

2. RELATED WORK

This section has the dual role of introducing model-driven engineering and circuit design automation to the readers as well as briefly presenting the state of the art in both fields.

2.1. Model-driven development

Model-driven approaches provides a methodology to tackle the complexity of software development using abstraction, problem decomposition, and separation of concerns. This methodology places models and transformations at the center of the software system development. A model can be seen as an abstract and simplified way to describe and understand complex systems. According to [12] a **transformation** is “the automatic generation of one(or more) target model(s) from one(or more) source model(s), according to a transformation definition. A **transformation definition** is a set of rules that together describe how a model in the source language can be transformed into a model in the target language”.

The massive adoption of the model-driven methodology was favored mainly by the standardization of Unified Modeling Language (UML) and Model-Driven Architecture (MDA) by the Object Management Group (OMG) [13]. Modeling languages such MOF, EMOF [14] or Ecore [15] as well as the development of environments like Eclipse Metamodeling Framework [15] have been largely adopted as the de-facto standard for model-driven software engineering. Efforts, like Kermeta [16], proposed ways to “breathe life into (your) metamodels”, by adding an executable meta-language, for operational semantics specification, to the contemplative (structural) side of model-driven development.

But the OMG is not the only actor improving on the state of the art in this field. The orthogonal classification architecture, proposed by Atkinson et al. [17, 18], provides a new insight into some possible future metamodeling environments, able to solve some of the current limitations.

The Software Composition Group proposes the use of Smalltalk as a “reflexive executable meta-language” [19, 20] for “meta-modeling at runtime” [11]. This approach addresses some more pragmatic issues around model-driven engineering, like the use of a single language – Smalltalk in this case – for all the modeling aspects, the possibility to dynamically change the meta-model at runtime, and the need to have executable meta-descriptions that alleviates the need for code generation.

In this study, we have adopted a hybrid approach based on the FAME meta-model [11] for domain modeling, and an ad-hoc Smalltalk-based behavior and algorithm specification approach, relying on the “transformation” metaphor, implemented using the visitor design pattern and isolated software components.

2.2. Electronic Design Automation

One of the principal problems addressed by electronic design automation (EDA) tools is the compilation of behavioral system specifications, in languages like C, Matlab to hardware systems [21]. From this perspective the EDA flow is typically decomposed in two compilation steps: high-level synthesis, and physical synthesis. The high-level synthesis (HLS) tools automatically convert behavioral descriptions of the system into optimized hardware designs, described using languages like VHDL or Verilog, which are then compiled to technology-specific logic netlists. The physical-design tools are used to map the technology-mapped netlists to electrically correct physical circuits.
In the context of this work we are looking at FPGA physical synthesis flow, which is composed of the following principal steps: partitioning, floorplanning, placement, and routing.

2.2.1. HLS UML extensions (profiles) such as SysML [22], Embedded UML [23], and MARTE [24] have been proposed for embedded system design. These profiles allow the specification of systems using high-level models that can be automatically (or semi-automatically) transformed to low-level hardware description languages, such as VHDL. In [25, 26], for example, the authors propose a HLS and design space exploration flow based on the MDE methodology, which based on successive refinements of a high-level model generates VHDL hardware accelerators.

2.2.2. FPGA Physical Design The Field Programmable Gate Array (FPGA) is a flexible computing architecture that bridges the gap between Application Specific Integrated Circuits (ASIC) and General Purpose Processors (GPP) by providing a customizable hardware fabric capable of implementing a wide range of applications. FPGA designs trade the performance of ASICs for flexibility and fabrication costs [27].

![Figure 1. The typical structure of a island-style FPGA.](image)

The basic structure of a FPGA is a regular mesh of logic blocks interconnected by a reconfigurable routing architecture. In the case of Island-style FPGAs the routing resources are evenly distributed throughout the mesh, the logic block usually having routing channels all around, see Figure 1. *

For FPGA physical-synthesis, a number of tools like VPR [28] and Madeo [1] aim at providing generic frameworks targeting reconfigurable architecture. They are based on high-level models, which ease the architecture and application description (based on specific DSLs [29]), and using generic heuristics (simulated annealing [30] - placement) and algorithms (Pathfinder [31] - routing) they automate the application mapping on the FPGA architectures. The principal limitation of these tools is the lack of support for software evolution: it remains difficult to extend them functionally by plugging-in new algorithms even though this remains a key feature to support the domain variability.

*In the scope of this paper when we say FPGA we are actually meaning Island-style FPGA
3. MADEO FPGA PHYSICAL DESIGN AND SOFTWARE DESIGN EVALUATION

Madeo [2] is a design suite for the exploration of reconfigurable architectures. It includes a modeling environment that supports multi-grained, heterogeneous architectures with irregular topologies. Madeo framework initially allows one to model FPGA architectures. The architecture characteristics are represented as a common abstract model. Once the architecture is defined, the Madeo CAD tools can be used to map a target netlist on the architecture. Madeo embeds placement and routing algorithms (the same as VPR [28]), a bitstream generator, a netlist simulator, and a physical layout generator. It supports architectural prospection and very fast FPGA prototyping. Several FPGAs have been modeled, including some commercial architectures (such as Xilinx Virtex family [32]), and prospective ones (such as STMicro LPPGA [33]). Based on the Madeo infrastructure further research projects emerged such as DRAGE [34], that virtualizes the hardware platform and produces physical layouts as VHDL descriptions.

The Madeo infrastructure has three parts that interact closely (bottom-up):

- **Domain model and its associated generic tools.** The representation of practical architectures on a generic model enables sharing of basic tools such as place and route (P&R), allocation, circuit edition [1]. Figure 2 illustrates MADEO on an island style FPGA. Specific atomic resources, such as operators or emerging technologies, can be merged with logic, since the framework is extensible.

- **High-level logic compiler (HLLC).** This compiler produces circuit netlists associated to high-level functionalities mapped to specific technology models. Leveraging object-oriented programming flexibility in terms of operators and types, the HLLC produces primitives for arbitrary arithmetic or symbolic computing.

- **System and architecture modeling.** The framework enables the description of static and dynamic aspects specific to different computing architectures, like: logic primitives, memory, processes, hardware-platform management, and system activity.

**Evolution of Madeo**

Not only the modeled domain moves fast, but it does so in an unforeseeable direction. As a consequence, some evolutions have led to abandoning features. The domain pressure was not
compatible with maintaining legacy artifacts which became obsolete over time. The symptomatic example comes from GUIs, that are often dedicated to a specific target. But more generally, reworking the model (migrating from 1-bit wires to composite wires as channels, bus, etc.) can have disastrous consequences (invalidating the visiting order, changing the textual description hence the compiler, etc.).

More delicate are the multiple-developer issues when some evolutions push - even without noticing it - a package out to the trash by breaking down its prerequisites. As an example, supporting conditions in the enumeration based HLS, prevents using a breadth-first visit of the graph; a depth-first is used instead, what means for any \( (a_i, b_i, ...) \) combination of the \( (A, B, ...) \) circuit’s inputs, outputs are produced and further propagated down into the child subgraph. This requires to store the current value of every output of every node, and was implemented by simply extending the model. However, as this kind of miscoding repeated over time, the model quickly became attribute-fat. One major branch of Madeo’s redesign was focused on improving the high level logic generator design through the use of trace-based analysis, with classical typing. However, that redesign effort simply ignored the domain modeling and physical design axis of Madeo, which is presented in this study.

The core reasons behind this decision were the inherent complexity of these tools and the limited time-frame allocated to that project.

The last point is that maintaining such a system, while still embedding legacy code from a time when almost no MDE existed, requires to keep in mind the whole picture; which is not scalable, neither in terms of personal investment, nor in terms of team work.

The re-design of the domain modeling and physical design axis of Madeo has been likely to be initiated for years. The integration of new members in the team has powered up the process. But, we have observed that the learning curve was too flat to educate non-Smalltalk experts to Madeo affordably. However through the redesign process presented in this study, we expect to render Madeo’s design cleaner and more robust which will likely conduct to a steeper learning curve.

Evaluating the Software Design of Madeo

To better understand this legacy code-base we have evaluated the physical-design subset of the Madeo toolkit based on a set of software engineering metrics, which can be classified into three distinctive categories:

- **size and complexity metrics** — the cyclomatic number\(^1\), lines of code, number of methods (# of methods), # of classes and packages, which count the principal entities in an object-oriented system;
- **coupling metrics** — the # of invocations, which represents the sum over all methods of the # of operations in the system calling each method, and the # of called classes. These two metrics characterize the coupling intensity and dispersion in absolute terms.
- **class hierarchy metrics** — The average number of derived classes (ANDC) and the average hierarchy height computed as the average of heights over all inheritance trees. The class hierarchy metrics show the usage of class hierarchies and polymorphism.

As agile programming promotes a “Test everything that could possibly break” approach in order to avoid some situations that will make the software misbehave, we apply a “Measure everything that could possibly degrade performances”.

Of course, an expert practitioner knows what is likely to break. He has the experience and the judgment to understand what this quote means in context. And adapting to context, as it turns out, is key to becoming an expert. Expert practitioners have enough experience to know what’s likely to happen next; and when it does not work out that way, they know what needs to change. It becomes apparent to them which plans need to be discarded and what needs to be done instead. Not only does this knowledge not exempt them to write tests but rather encourages them to increase the number and relevance of the tests.

\(^1\)The cyclomatic number is the sum of the total number of possible paths of each method [35]
Table I. Statistical thresholds for 20 Smalltalk systems computed to create a baseline for evaluating Madeo physical-design

<table>
<thead>
<tr>
<th></th>
<th>Low</th>
<th>Average</th>
<th>High</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Intrinsic Operation Complexity</strong></td>
<td>1.16</td>
<td>0.22</td>
<td>0.27</td>
</tr>
<tr>
<td>Operation structure</td>
<td>4.92</td>
<td>6.91</td>
<td>8.90</td>
</tr>
<tr>
<td>class structure</td>
<td>8.18</td>
<td>11.24</td>
<td>14.30</td>
</tr>
<tr>
<td>high level structuring</td>
<td>4.63</td>
<td>8.57</td>
<td>12.50</td>
</tr>
<tr>
<td>coupling intensity</td>
<td>3.13</td>
<td>5.46</td>
<td>7.79</td>
</tr>
<tr>
<td>coupling dispersion</td>
<td>0.15</td>
<td>0.23</td>
<td>0.31</td>
</tr>
<tr>
<td>Average # of derived classes</td>
<td>0.37</td>
<td>0.60</td>
<td>0.83</td>
</tr>
<tr>
<td>Average hierarchy height</td>
<td>1.42</td>
<td>2.22</td>
<td>3.03</td>
</tr>
</tbody>
</table>

However, not all programmers are experts. Tests offer executable specifications acting as a guideline to them, and exhibits the behavior of a faulty application. Besides, it prevents regression during evolution. Therefore, test-driven development is a safe, durable, and affordable way of creating software.

Our metrics-based approach conforms to a similar scheme. It provides a baseline, against which we measure any refactored version. Besides, metrics spread out over a wider range, with no pre-oriented results, so that any unforeseen impact is unlikely to remain undetected.

We - as domain experts - do not need these indicators prior to redesigning Madeo. Instead, we can learn from the evolution of the results that the metrics provide over time. Our expertise grows as we start isolating some correlations between these results and internal good practices.

As TestRunner executes all tests, our tool measures all metrics. As testing should remain fast and affordable in terms of coding, our metrics should be scored quickly hence often and require no extra development as the Moose re-engineering framework embeds all required facilities.

Not all of the lessons learned are potentially applicable to any other Smalltalk project as Madeo exhibits some specificities. However, applying metrics offers a sound basis for evolution scoring, for trying to understand unexpected changes in performances, and can further provide some design rules to domain experts.

The first two metric categories provide absolute values characterizing the system under study. To extrapolate the results and get any insight into the different aspects of the system the absolute values obtained are combined into comparable ratios enabling the comparison with other projects. Grouped by the initial categories, these derived metrics are:

- **size and complexity ratios:**
  - Intrinsic operation complexity, the number of conditional branches per lines of code;
  - Operation structure, the number of lines of code per methods;
  - Class structure, the number of methods per class;
  - High-level structuring, the number of classes per package.

- **coupling**
  - Coupling intensity, the number of invocations divided by the number of methods;
  - Coupling dispersion, the number of classes divided by the number of invocations.

The obtained results were compared to Smalltalk-specific statistical thresholds obtained by computing these same metrics for over 20 different Smalltalk systems (varying in size and complexity). Table I shows the obtained results, which offer a comparison baseline. To compute Low and High threshold values we have used the difference between the Average and the standard deviation ($AVG - STDEV$) and, respectively, the sum ($AVG + STDEV$).

The results obtained are presented in Figure 3 using the OverviewPyramid graphical representation proposed by Lanza et al. in [35], which shows the three different metric categories as a pyramid containing the absolute results as well as the derived metrics. Figure 4 shows the results obtained for our legacy system (gray area) compared to the Smalltalk design thresholds to get a...
visual perspective of the amount of deviation (in % — 100% represents the average threshold) from a typical Smalltalk project.

Figure 3. “Overview Pyramid” of physical-design sub-system of Madeo. The red/blue rectangles represent results that are closer to the high/low statistical threshold

Figure 4. Comparison of Madeo physical-design sub-system results with Smalltalk specific statistical thresholds

From the results obtained on the size and complexity axis, we can infer that our legacy system compared to an average Smalltalk system has a coarse-grained package structure with large classes composed of rather large and relatively linear methods. The large package structuring, can be seen as a development policy which makes sense in the case of a physical design framework which intrinsically is structured around two large models which can be isolated into two different packages with the third package, also coarse-grained, containing the implementation of all physical design heuristics. However, the large number of methods per class (over 60% bigger than the high threshold) and the large methods (over 20% higher than the threshold) are a clear indication of the need for refactoring to split large classes with unrelated methods into smaller entities, and to refactor the large methods into smaller ones that delegate certain operation to others. The low intrinsic operation complexity can be seen either as an indication of code duplication, as a domain-specific characteristic — consequence of a large domain-model with many simple accessor methods, or simply as an outlier consequence of the existence of methods containing binary-logic circuit
examples. In either case this again indicates the need for further evaluations to better understand the system to be able to propose adequate refactorings.

From the results obtained on the coupling axis, we observe that the system is intensively coupled in terms of operation calls, but these calls are very localized, i.e., functions tend to call many operations from few classes. These characteristics can be seen either as an indication of the existence of god classes or as domain-specific since the two large domain-models (application and architecture) do not implement complex algorithms but they outnumber certainly the few physical-design automation classes which implement the core platform algorithms. If this is the case, the very large coupling intensity number (almost 40% bigger than the high threshold) indicates that the two domain models hardcode the references towards the algorithmic classes which has a negative impact on software evolution (e.g. it is hard to integrate new algorithmic classes).

The evaluation of the class hierarchy metrics shows that the system frequently uses deep class hierarchies, which in our case is clearly a domain-specific characteristic since the analyzed code-base contains two distinct deep hierarchies that represent roughly 80% of the number of classes, and which is used for reifying the domain concepts (application and architectural concept).

By corroborating these independent results we can conclude that though the physical-design subset of the Madeo framework makes extensive use of deep class hierarchies compared to a typical Smalltalk project it poses a number of interesting questions from a software engineering perspective. Moreover, based on the large values (well exceeding the high threshold) obtained for 4 of the metrics we observe the existence of potential problems on the size and complexity axis in relation with the high coupling intensity observed.

The following sections of this study present a practical approach to structurally refactor this framework to solve some of these issues and finally to evolve this legacy code-base towards a new toolkit based on model-driven engineering concepts.

4. MODEL-BASED EDA IN SMALLTALK

Twenty years ago the executing environments provided by virtual machines was considered too slow for solving the hard combinatorial optimization problems of circuit design. But, industrial successes such as the ControlWorks [36] wafer fabrication platform, coupled with research efforts such as Programmable Active Memories [37] (PAM), and the MADMACS [38] layout editor inspired the community and provided the incentives to use interpreted languages for circuit designs. Approach that trades-off some of the performance advantages of close-to-the-metal languages to the ease of use, the expressiveness, and ultimately the undeniable productivity of high-level languages.

Amongst the numerous efforts in this direction, the Madeo project [1] has overcome the performance challenges of object-oriented languages in the context of FPGA design automation. It showed that dynamically typed languages can serve for creating high density logic designs [39, 40], proposing the first virtual FPGA prototyping environment [41] and proving that, harnessing the power of OO software design, one could create a flexible yet competitive EDA toolkit that even today enables breakthrough research in the field [42, 34]. Meanwhile OO design became widely accepted by the EDA community through languages like C++ (i.e. systemC [43]) and Java (i.e. JHDL [44]). But OO design suffers from a number of problems, common in software engineering, especially from a software evolution perspective. As the target domain evolves the software tools used for modeling, simulation and reasoning about the domain have to evolve accordingly. And without a methodological approach for evolution, systems become unmaintainable and, even more, functionality is lost through numerous integration cycles. Model-driven engineering promises a

---

‡A god class is a large class implementing many operations, in object-oriented software design they are considered anti-patterns.

∗ControlWorks was developed by Texas Instruments using the Smalltalk environment.

†Programmable active memories reified the application structure enabling circuit generation from a high-level object-oriented language.

‡MADMACS proposed the use of a functional language (similar to Lisp) for creating VLSI layouts.
Table II. How are the parsers implemented into the Madeo framework.

<table>
<thead>
<tr>
<th># of parsers</th>
<th>T-Gen</th>
<th>Smacc [49]</th>
<th>Hand-written</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>3</td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

number of solutions for this problem by abstracting and decoupling the different software artifacts, most notably through the use of aspect-driven programming and component-based design. In consequence, today we assist to yet another step in the evolution of EDA industry, the move towards model-driven and component based design, especially in the context of embedded system development and high-level synthesis.

During the last years Smalltalk world has undergone numerous evolutions, through open environments like Pharo [45], language innovations such as: the adoption traits for fine-grained reuse [46], language boxes for embedded DSLs [47], and runtime meta-modeling frameworks like FAME [11]. Moose project experience report [20], as well as our experience with Platypus [48] led us to create an agile MDE-based prototype, targeting FPGA physical-design automation, using the Smalltalk environment. Our goal is to create an evolution-aware platform, relying on our legacy code-base, that can evolve to accommodate new technological requirements.

The most important aspects that need to be addressed by such a toolkit are: domain modeling, domain-specific languages, code reuse, legacy and fast prototyping, and external tools integration. In the following paragraphs we will review each of these aspects and briefly present some Smalltalk technologies that can address each issue.

Domain modeling  is at the core of any EDA toolkit. In the context of rapidly changing IC technology, the domain model has to evolve rapidly, eventually changing its internal structure. For addressing this problem the FAME meta-modeling library [11] proposes a solution by offering runtime access to the meta-information, appearing in a meta-model. By creating a bidirectional causal connection between the meta-level and the implementation-level the model updates can be reflected in-between the two levels at runtime. In Section 4.1 we will present in detail a FAME-based abstract meta-model that is used throughout our framework as a common base for creating the different domain models needed.

Domain-specific languages  are used extensively in the context of circuit design automation. Their principal roles are: describing the IC architecture (Hardware Description Languages (HDL)), describing the different physical, technological, and design constraints, describing the functional requirements, specifying test cases and objectives, describing the application netlists, etc. In the context of our framework, we have implemented more than a dozen different parsers for interfacing purposes and we have a proprietary HDL used for FPGA architecture description and instantiation. Table II shows the different approaches used for implementing these parsers. Hand-written and table-based parser descriptions, such as T-Gen or SmacCC-based [49], are difficult to modify according to the evolving needs of the EDA field. The parser-combinator libraries such as PetitParser along with the language-box concept implemented in the Helvetia DSL development environment [47] provides a new solution to this problem. Helvetia enables the seamless creation of embedded DSLs, while PetitParser brings inheritance to grammar specifications, thus offering the possibility to isolate the grammar specifications from the abstract syntax tree creation. These developments provide a smart way for defining a concrete text-based syntax for the instantiation of our domain models. It should be noted that due to the high number of hardware modules included into a typical architecture, visual languages are not well suited for the architecture description. Moreover IC designers are very proficient using textual description languages.

Code reuse  is the most important concern from a software evolution point of view. OO methodology provides inheritance and polymorphism as de-facto solutions for enabling large-scale code reuse. With the adoption of traits [46], in Smalltalk dialects such as Squeak and Pharo, the
OO toolbox for reuse gained a new tool. Traits are a mechanism for fine-grained method reuse that decouples functional method implementation from the object state.

**Legacy and Fast Prototyping.** Any evolution methodology has to enable incremental development, ensuring the usability of the environment during the whole process. For this study, the prototype was bootstrapped by reusing the MADEO project infrastructure, and then it has incrementally moved towards our new MDE-based framework. In Section 7 some of the evolution steps are presented to better illustrate the approach.

**External tools integration** is a high-level reuse mechanism by which tools developed by third-parties can be integrated into the toolkit. Historically we have been using inter-process communication (via UnixProcess class) and FFI (such as DLLCC) for executing external programs and interfacing with our toolkit. Moreover, the transformation metaphor, proposed for algorithm design, opens the toolkit even more by enabling a fine-grained external tool reuse, via the *External* atomic transformation.

The remainder of this section presents our practical approach for structural domain modeling and a novel conceptual framework for designing and implementing algorithms based on the transformation metaphor.

### 4.1. Fame-based Domain Modeling

From a modeling point of view most of the EDA tools are structured around hierarchical models appearing at different levels of abstraction. These models are used to represent the two principal axes of EDA:

- **Application.** The applications are typically seen as a composition of operators, which can be further broken-down into simpler constructs towards elementary systems primitives. The behavior specifications can be treated at different abstraction levels: i.e. control data flow graphs, with hierarchies representing processes and/or different functions used to implement the application, combinatorial or sequential logic, used to describe the system behavior as composition of simple Boolean functions, etc.

- **Hardware.** The hardware designers rely heavily on hierarchical descriptions to simplify the integrated circuit development. As for the applications, the hardware is described hierarchically at different abstraction levels. At the system-level, for example, an IC can be seen as an array of blocks, each of which implements a specific functionality (i.e. processors, GPUs§, network modules, etc.). Each of these blocks can then be decomposed into their own building blocks (memory, logic, interconnection, etc.). At the logic-level, a digital circuit can be seen as a composition of logic blocks, implementing different functions according to the Boolean equations characterizing the blocks. At the circuit-level the system is again specified hierarchically, using transistors, diodes, etc. as primitives.

Based on these observations, we created an abstract meta-model that is used to structurally describe our domain as a hierarchical composition of primitive elements interconnected together. This abstraction describes mainly a hierarchical annotated port-graph. Using the FAME metamodeling framework it was straightforward creating a Smalltalk implementation of this high-level domain-specific modeling language.

In the proposed meta-model, see Figure 5, the domain structure is captured as a *Composite* that contains one or more *Entity* instances (called “entities” from now on). *Entity* is an abstract construct specialized as *Leaf* and *Composite*. The *Leaf* instances are domain primitives which are viewed as the indivisible building blocks of more sophisticated structures. The entities have hooks (*Hook*) which provide an external interface through which they can be interconnected together. The abstract *Hook* concept is specialized as *Pin* and *Port*. The *Pin* element allows connection between entities.

---

§ graphical processing unit
It can be a physical contact, or a logical interface depending on the specification level or on the refinement degree. The role of Port instances is to group together and structure the Pin instances, they can be seen as namespaces structuring the access to particular pins. The Connection purpose is to glue together the entities of a particular system by linking pins together. These connections can be interpreted as wires, communication channels, or simply as relations between entities.

All these concepts are refinements of the Class element which owns Property instances. The Property represents different attributes particular to the modeled domain. Some examples of properties are: the position of an entity on a layout, the capacity or resistance of a wire modeled as a Connection instance, the logical value of a circuit pin represented as a Pin instance.

The meta-model, presented in Figure 5, represents a common abstraction which can be used to specify any kind of interconnected system having any number of hierarchical components that present the system at different abstraction levels. This core structure is refined in Section 5.2 to describe an island-style FPGA architecture and in Section 5.3 to capture the characteristics of a combinatorial circuit.

Figure 6 shows a meta-diagram (visualization at the abstract level presented in this section) of a composite block modeling an inverter chain. The use of this kind of meta-diagram enables quick visual verification of the modeled component without having the need to create a specific viewer. The principal drawback is that while generic, this abstract viewer will generate a large amount of visual artifacts, which in some cases will obscure important details. Nonetheless, these meta-visualizations are useful especially during the first prototyping phases of the refined models.
4.2. Transformation Metaphor for Algorithm Design

Besides structural domain modeling, algorithm design is the most important aspect of any EDA toolkit. Since almost all the optimization problems encountered in electronic CAD tools are NP-hard in terms of complexity, most of the practical solutions rely on heuristics. Hence, the main concern of EDA tools users and designers is the heuristic performance in terms of execution time, memory requirements, and solution optimality. It is commonly accepted for an EDA tool to run for days or weeks on high-end machines having huge memory resources. In terms of implementation, these optimization heuristics are very complex. Most of them are implemented in highly optimized C, for performance issues.

The Madeo framework itself relies on external tools, mainly for logic synthesis and technology mapping. But it implements generic heuristics for the physical design problem to assure the flexibility of the mapping. The idea is that these generic heuristics can be used in the context of different FPGA architectures with the minimum human intervention for optimization goals parametrization.

To address the algorithm design problem in this study we propose a technique that we call “Transformation metaphor”. This technique appears as a conceptual framework; the algorithm designer now looks at the algorithm implementation as it was a model transformation problem.

This approach mainly reifies the implicit tight dependency between algorithms and domain models, through explicit transformations, isolating their respective concerns, thus increasing the flexibility and the expressiveness of our toolkit.

In the transformation metaphor each algorithm (or heuristic) is seen as a hierarchical composite transformation. Figure 7 presents the different types of transformations proposed by this approach. The primitive transformation types are: concern extraction, and atomic transformations.

The concern extraction represents the mapping from a domain model to simpler models required by the algorithm being implemented. The purpose of the concern extraction is to decouple the domain specific details of the model from the algorithm-specific details.

From an implementation perspective, the concern extraction is nothing more than implementing a visitor (according to the visitor design pattern [50]) that will iterate over the domain model extracting information and instantiating an algorithm-specific model.

The atomic transformations represent actual algorithms (or heuristic) needed to solve the problem. According to the specific needs it can further decomposed in more fine grain modules and composed as using the transformation metaphor or it can be directly implemented in a programming language on demand. The only requirement is to export clear interfaces so that it can be integrated as an atomic transformation in the framework.

This approach has the advantage of being able to integrate external tools implemented using any programming language and/or computing model. To better illustrate this approach Section 6.2
presents the design of a FPGA routing algorithm as a practical example, and Section 6.3 presents the results obtained in terms of development cost, performance, and extensibility.

Figure 8 shows a typical FPGA physical synthesis flow. It starts from the description of a particular FPGA architecture as an architectural model (ArchM) instance, and a target application as an application model (AppM) instance. These models are domain specific, fully independent from any algorithms that are used for physical design. The output is a refined ArchM instance, configured to implement the AppM instance.

Despite this flow is domain specific, it exhibits similarities with distributed programming: as a comparison, let’s consider a software application to be executed over a cluster of workstations. The application must be compiled (mapped onto processors ISA), as the partitioning step ensures the refactored application fits the technology-mapping constraints. Threads are clustered in order to perform load balancing, what means coarse allocation is carried out to minimize some metrics such as execution time or interconnection extra-cost. Similarly, Floorplanning allocates zones in the FPGA, so that the whole area is minimized with regards to interconnection. Running threads on a processor requires scheduling; in other words, a fine grained resources allocation (time quantum as processors execution conforms to a temporal scheme) is performed. Placement also appears as a fine grain allocation (logical cells within a zone, as the execution is spatial). Once done, interconnections are handled as either pipes, remote connection, etc. for processors or routing resources configuration (last stage) for FPGAs.

![Figure 8. A standard physical design flow. The rectangles represent models, while the ellipses represent algorithms.](image)

5. MODEL-DRIVEN DEVELOPMENT FOR FPGA PHYSICAL DESIGN

The purpose of this section is to show how the Smalltalk-based MDE methodology (presented in the previous section) is used in the context of the FPGA physical design.

The physical design step is responsible for allocating all design components for creating the configuration for FPGAs (bitstream). This means that each gate of the application netlist will be assigned a spatial location (placement). Then, the interconnection signals will be reified using appropriate routing structures. Physical design has a direct impact on the circuit characteristics (performance, area, power, etc). The main steps of physical design are: partitioning, floorplanning, placement, and routing [51].
5.1. Tool-flow Modeling

The flow is an endogenous transformation realized via four composite transformations (see Figure 8): partitioning, floorplanning, placement, and routing. Each of these four transformations is a composition of more-elementary transformations. Thus, the physical synthesis tool flow is a hierarchical directed acyclic graph (DAG) of transformations, where the nodes represent the transformations to be done, and the edges represent the dependencies between the tools. The execution of the transformations happens in topological order from the DAG inputs (ArchM and AppM instances) to the outputs (AppM instance mapped on the ArchM instance).

The principal advantage of using model-to-model transformations for tool-flow modeling is the capacity to easily replace any of the physical synthesis algorithms with different implementations, with no constraint on the implementation language or formalism. But this also has a drawback, the high number of simple transformations (concern extraction) needed to implement the flow. However, such a view of the physical design automation, that isolates and encapsulates the different steps of the flow, poses the bases for future standardization, which that can cut down the number of intermediate transformations.

In the following paragraphs we introduce two different specializations of the hierarchical port-graph model, introduced in the Section 4.1. Then in Section 5.1 we present the tool-flow modeling methodology based on the “transformation metaphor”, used for implementing the four steps of the FPGA physical design flow.

5.2. Island style FPGA model

To capture the particularities of FPGA architectures (see Figure 1), the meta-model, presented in Section 4.1, was refined. Figure 9 shows the domain-specific concepts added.

The integration of this abstract common representation into the Madeo framework enabled the reuse of otherwise duplicated concepts at the FPGA modeling and Application modeling layers. Moreover, it offers a common infrastructure for architectural and application specification that is based on one common domain-specific language abstraction, which is then refined to target these two specific modeling layers. Practically by using this model we removed 6 duplicated top-level domain concepts: the primitive objects (MH.Leaf), the composite entities (MH.Composite), the pins (MH.Pin), the references to pins (MH.PinAlias), the ports (MH.Port), and the connection entities (MH.Connection).

Using this meta-model the FPGAs are modeled based on 3 primitives (specializations of MHLeaf): Logic Block, Switch, and Connection Block. LogicBlock instances represent the programmable logic blocks which provide the basic computation and storage elements. The routing architecture is modeled using Switch and ConnectionBlock instances. These primitives contain a number of internalConnections that specify the way their pins are connected internally (see Figure 10). Tile instances aggregate these primitives into logical bricks which are replicated regularly to form a Cluster. The primitive pins are aliased at the tile level and cluster level, using InterfacePin instances, to expose the possible connection points.

Figure 10 presents a view of the principal types of internal connections. HardConnect can be seen as a direct connection between components without any physical property (it is used for example to represent long wires which logically are composed of different wire segments, the connection between these wire segments is a HardConnect instance). Wire represents a physical wire. ProgrammableConnect is an abstract representation of electrically configurable components.

The FPGA meta-model (ArchM) presented in this section is not intended to exhaustively cover the programmable architecture design concepts, nor to be used as a real-life FPGA meta-model but rather is a simplified version that captures the basic properties of interest for the purpose of this study.

5.3. Combinatorial Logic Meta-Model

In this section we present another refinement of the meta-model, presented in Section 4.1, focusing on hierarchically description of combinatorial logic.
Figure 9. A simplified view of the Island-style FPGA extension of the core meta-model

Figure 10. Connection hierarchy

In digital electronics, combinatorial logic is a type of digital logic implemented by Boolean circuits where the logic network computes a pure function based solely on the inputs. As opposed to sequential logic that uses the inputs and stored previous state variables for computing the outputs.

Figure 11. Combinatorial Logic extension of the core meta-model

The combinatorial logic meta-model (AppM), presented in Figure 11, uses two-level single-output logic functions (LogicGate) to model the application netlist.

As for the FPGA model, presented in the previous section, only a simplified view of Combinatorial Logic meta-model is presented. Section 6 presents the details of the FPGA routing step viewed as a composite transformation.
6. CASE STUDY – ROUTING ALGORITHM

This section illustrates the approach and its benefit through addressing a critical task in the FPGA programming flow: routing. Routing exhibits on one hand a high complexity, as the number of paths between two pins in an architecture is highly sensitive to the Manhattan distance between the source and the sink. On the other hand, the algorithm can be optimized by considering the target specification.

As a result, routing is a major variation point. A baseline is provided by the VPR tool [28] and the Pathfinder algorithm [31], but every hardware architect wants an optimized router for his architecture.

Supporting routing as a composite transformation is a milestone in building up a domain-space-exploration feature for algorithms. As target are increasingly heterogeneous, one algorithm is unlikely to fit efficiently to all parts. Another beneficial effect is to serve a divide-to-conquer approach well.

6.1. Problem Definition

The FPGA routing architecture can be modeled with a routing graph \( G_r(V_r, E_r) \) [31, 28], which is a directed graph. The vertex set \( V_r \) represents the input and output pins of logic modules, and the wire segments. The edge set \( E_r \) represents the possible connections between the nodes. The nets to be routed are directed hyper-edges on \( G_r \), \((s_i, t_j^i)_{1 \leq j \leq n}\), where \( s_i \) and \( t_j^i \) represent the source, and respectively the sinks of the net \( i \).

A net routed corresponds to a subtree of \( G_r \), called the routing tree for the net. The root of the routing tree corresponds to an output pin (in the FPGA architecture) and is the source of the net. The leaves of the tree are input pins of logic blocks, and correspond to the sinks of the net. Since an electrical path cannot be shared by different signals the routing trees for the nets should be vertex disjoint, this is called the exclusivity constraint [52].

The routing problem can be stated as finding vertex disjoint routing trees in \( G_r \) for all nets while satisfying performance constraints.

In this section we consider a timing-driven router based on the Pathfinder algorithm [31]. For timing calculations we use Elmore delay model [53], that uses the architectural parameters to compute the source-to-sink delay. In timing-driven routing, the timing constraints are specified as arrival time \( (A_t) \) and required time \( (R_t) \) at primary inputs or outputs of storage elements. A timing graph, \( G_t(V_t, E_t) \), is created from the placed application netlist to represent these information.

The goal of the timing-driven routing problem is to route all nets such that the delay on the critical path is minimized while satisfying the delay and exclusivity constraints.

6.2. Routing as a Composite Transformation

From the MDE perspective, the routing problem is a refinement of the partial ArchM instance. This refinement consists in the identification of the routing paths in the ArchM instance for the nets from the AppM instance. The transformations required to solve the routing problem are classified in 3 categories: pre-routing transformations, core routing, and post-routing transformations.

The pre-routing transformations are a preprocessing step that identifies and extracts the elements directly needed by the core routing transformation, \( G_r \), \( G_t \), and the nets in our case. The pre-routing transformations are simple transformations, concern extraction (as we called them in the last section), that can be implemented using transformation engines.

The RRGraph extraction transformation creates the routing graph \( G_r \) from the ArchM instance. It also computes the delay for the possible connections by applying the delay model on the RC parameters of the ArchM instance. The nets are extracted from the AppM instance using the placement results to identify the architectural elements corresponding to source and sinks. A partial instance of the timing graph is created \( G_t \) from the placement results and the AppM instance. We call \( G_t \) a partial instance because the delays of the routing paths between the logic blocks are not known, being computed during the core routing transformation.
The core routing transformation is a composite transformation (see Figure 12) of the main algorithmic aspects of the routing problem. For the timing-driven pathfinder algorithm these aspects are: signal routing, and timing graph refinement. If these aspects are treated as transformations, they can be implemented apart from the core algorithm, thus decomposing the routing problem event further. The principal roles of the routing, in the case of Pathfinder, is to produce the routed nets according to the timing and exclusivity constraints, to refine the timing graph model, by adding the cost of the routing paths, and to update the costs of the routing resources in \( G_r \), since the exclusivity constraint is solved by resource negotiation.

Once the core-routing transformation done, the post-routing transformation refines the ArchM instance injecting the routed nets in the model. The ArchM instance obtained represents the solution of the physical synthesis flow, the mapping of the AppM instance on the architecture.

### 6.3. Results

To study the impact of using model-to-model transformations for the design of physical-design heuristics this section explores and quantifies three different aspects related to the implementation of the routing automation step, presented in the previous section:
Table III. The cost of architecture-specific routing-cost integration in the legacy and the model-driven toolkits

<table>
<thead>
<tr>
<th></th>
<th>Legacy</th>
<th>Model-driven</th>
</tr>
</thead>
<tbody>
<tr>
<td>modified classes</td>
<td>14</td>
<td>0</td>
</tr>
<tr>
<td>modified methods</td>
<td>23</td>
<td>0</td>
</tr>
<tr>
<td>new classes</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>new methods</td>
<td>0</td>
<td>20</td>
</tr>
</tbody>
</table>

- Firstly, the development cost of integrating architecture-specific characteristics into the toolkit is discussed, with an emphasis on the extent to which the default toolkit functionality is modified.
- Secondly, the impact of decoupling the optimization data-structures from the architectural model is evaluated with respect to the MCNC application benchmarks [54]*.
- Lastly, the extensibility of the approach is emphasized by overviewing the different routing policies integrated in the system.

Table III presents the number of classes and methods modified and/or added to the system to adapt to an architecture which has a different routing-cost characterization compared to the default policy hard-coded in the Madeo system. Historically, for the Madeo toolkit, this implied the modification of up to 23 methods in 14 classes. These modifications are applied at different points in the architectural model hierarchy. At the beginning of the project the main objective was to rely on inheritance to spread these values down the hierarchy. However, during the toolkit lifetime we have observed that certain methods get duplicated for particular architectures and that finally they get integrated into the main development branch. Thus, invalidating certain predefined use-cases and rendering the system unstable. In consequence, the system got difficult to use and maintain, and the system users were pushed towards the creation of architecture-specific development images which further increased the difficulty of version integration. The use of the algorithm-as-transformation metaphor directly addresses this problem by the use of concern extraction transformations. These transformations, currently implemented using the Visitor design pattern [50], isolate these variability points from the model, and structure them around an aspect-specific transformation. In the case of the architecture-specific routing-cost, presented in Table III, the user has to create only one new architecture-specific class and overwrite up to 20 methods to achieve the same result as in the previous case.

The memory overhead is the main cost of decoupling the architectural model instance from the physical design automation steps since an algorithm specific model will be constructed in memory by the concern extraction transformations. The impact of this new model instance on the total memory consumption varies according to the amount of information needed, and typically increases as the mapping of the application gets more and more refined toward the end of the automation flow. After measuring the memory usage of our automation flow and comparing the results with the memory usage of the standard Madeo environment, we have observed that during floorplanning and placement there is no visible difference in memory consumption since the Madeo environment itself is instantiating internally floorplan and placement specific objects. However, the approach presented in this work has a negative impact on the memory consumption during the routing step. The main reason being that the ArchM instance already is an implicit routing resource graph that, in the case of Madeo, is annotated with timing information and algorithm specific attributes. However, for our approach two different graph models will be extracted from the architectural model and instantiated in memory.

Figure 13 presents the memory overhead of instantiating the routing-resource and timing graphs depending on the benchmark application (x axis). Besides the percentage ratio of memory used, this figure shows that the overhead is strongly correlated (0.86 correlation coefficient) with the size of

---

*The MCNC benchmark set is a compilation of circuit designs largely used for measuring CAD algorithms performance in terms of the quality of the results, the execution speed, and memory footprint.
the architecture in terms of modules\footnote{These results were obtained based on applications from the MCNC benchmark set \cite{54}, which are placed and routed on architectural instances matching each benchmark size (in terms of modules). The amount of routing resources needed for successfully routing was computed dynamically during a preprocessing step, which relies on a simplified version of the router presented in this section.}. In the case of misex3 the memory overhead is much greater than in the other cases mainly as a result of the high connection density between the logic blocks of this particular application, which demands an architecture with a large number of routing resources for assuring successful routing. Hence the extracted routing resource graph in this case has more nodes and edges as compared to an application with low connection density.

On the studied MCNC benchmark we have obtained a 22\% average memory overhead for instantiating the routing specific data-structures (routing resource graph, and timing graph) needed for implementing the timing-driven Pathfinder router \cite{31}, presented in this section. While this number is fairly large and varies between less than 5\% to over 40\% in certain cases it also shows a positive aspect of our approach, mainly that by decoupling the architectural model from the flow we can potentially get up to 78\% smaller memory usage during physical design by storing the architectural model in an external database. A reasonable objection to this claim is that in order to traverse the model, one needs to strictly maintain a wide part of the model in memory, hence that this gain remains theoretical. However, this is not true, as required information can be restricted across two axis: first in space, as only a portion of the architecture area is allocated while implementing the circuit, and secondly in a structural direction as only relevant information is required at a time. Our approach supports caching mechanism with possible load/unload clever policies (eg. advanced prefetching to ensure data are available on time, since the sequence of tasks is known from start) that are beyond the scope of this paper. Moreover, the routing graphs can be optimized for space by compacting recurring patterns \cite{55}.

Furthermore, to validate the viability of the algorithm-as-transformation approach four different routing strategies were implemented and the recurring software patterns were refactored into an external routing framework which can be used independently from the underlying architectural model. In this context, the studied FPGA routing policies were:

- the timing-driven pathfinder router \cite{31} described in this paper;
- a routability-driven router based on the flow in Figure 12, but without the timing optimization specified by the TGraph Extraction, AT Refinement and RT Refinement transformations;
- a lagrangian-relaxation based timing-driven router \cite{56};
- the pathfinder-based router implemented in the VPR tool \cite{28}.

In the case of the VPR tool, the pre-routing transformations create an instance of the VPR architectural model using the XML-based description language, and a netlist using the .net file.
format needed by the VPR tool. The post-routing transformation reads the routing results file and injects the routes in the initial ArchM model instance. This shows that core-routing transformation, the Pathfinder router in the example, can be easily replaced by any other domain independent tool having the required functionality. Moreover this result is valid for all the other core algorithms needed in the physical synthesis flow, presented at the beginning of section 5.

7. FROM LEGACY TO MDE TOOLKIT – SUCCESSFUL EXPERIENCES

This section provides an account for some of the steps we followed towards the MDE-based framework. Moreover it shows the flexibility of the approach as the environment remained usable during the whole evolution process.

7.1. Improving on Legacy – First steps

The first modifications that were integrated in the toolkit focused on improving some of the optimization routines already present in our legacy code-base, namely the floorplanning and routing routines.

In the case of the floorplanning routine, we have chosen to replace the TCG-based [57] heuristic present by an improved version relying on a different floorplan representation, namely TCG-S [58]. From the implementation perspective we tried to decouple as much as possible the heuristic from our domain models, so that it can be reused in other contexts with no modifications. The integration into the toolkit was done by redirecting the automation flow towards the newly created module. Concern extraction was used to instantiate the TCG-S specific floorplan model from the AppM using ArchM geometrical information. Once the floorplan model instantiated, the optimization goals (metrics) were added as closures (Smalltalk blocks) independent of the heuristic implementation.

For the routing routine we refactored the existing routing algorithm (Pathfinder [31]) decoupling it from the architectural model with which it had numerous dependencies, and we created a transformation-based version. As for the TCG-S algorithm, the architecture specific optimization goals are set using closures. The results using this new implementation were over 40% faster than the initial implementation, mainly due to the possibility to prune the routing resource graph, thus reducing considerably (≥50%) the number of nodes explored during the execution. One negative aspect of this approach is the increase in the memory footprint due to the duplication of some ArchM aspect in the routing specific model.

7.2. Extensions for Nanoscale Physical Design

In [3, 4] the extensibility of the MADEO framework was put to a test for the first time with the advent of emerging technologies. The core concepts of the NASIC fabric [59], see Figure 14, were introduced into the framework, and a reconfigurable nanoscale architecture, called NFPGA, was designed. This required to extend both the reconfigurable architecture model and its associated tools in such a way that NASIC can be modeled and programmed. Process that goes through several steps:

1. The generic model must evolve to describe new potential components (nanogrid, nanowire, etc.) with specific characteristics.
2. This generic model must be instantiated using a proprietary HDL. As the HDL expresses the model, any in-depth change within the model leads to an evolution of the HDL (i.e. new keywords).
3. Some algorithms must be adapted or substituted for technology-specific counterparts while preserving the API. For example, the logical functions are implemented using a 2 level logic rather than FPGAs LUTs or processor μ-instruction.

More recently [60, 61] the methodology presented in this study was used to propose a complete physical synthesis tool-flow for a new nanoscale architecture template. The legacy Madeo
toolkit was used for placement and routing (as well as for archM description, instantiation and visualization), external tools like Sis [62], PLAMap [63] were seamlessly integrated with new internal tools for PLA-family‡ exploration, and metric computing. Different tool-flows were created using these tools, each one having different optimization goals, and working on different architecture variants. Moreover by opening the toolbox the design-space exploration (DSE) was bootstrapped relying on standard reconfigurable place & route routines, thus enabling a baseline evaluation which showed the need for more optimized routing. Once the new routing algorithm was developed it was integrated into a new tool-flow, specializing the baseline tool-flow via inheritance.

The main conclusion of this experiment is that using this MDE approach, not only is effective incremental design-space exploration back, but a new tool-flow exploration axis is added to the typical application/architecture trade-off. Besides the tool-flow specialization reduces the development effort.

7.3. Refactoring Domain-Models

The most extensive evolution of our legacy code-based was the replacement of old domain-specific OO models with a newly engineered set of FAME-based domain models, relying on the hierarchical port-graph abstraction, described in Section 4.1. The preservation of legacy functionality was the principal constraint in this case. To address this problem engineered replicas of the legacy model entities were merged into the new model hierarchy to factor the available functionality of the two. Two different automated merging policies were devised: copydown method that destructively change the inheritance hierarchy of the legacy model to attach its entities to the new hierarchy; and doesnotunderstand method that detours the execution call-flow towards the legacy model classes by intercepting the #doesNotUnderstand Smalltalk message in case of missing functionality in the new hierarchy. Both of these merging policies are extensively presented in [10].

‡Programmable Logic Array
7.4. Evaluation of the Model-Driven Prototype

To put these experiences in perspective, we performed the evaluations presented in Section 3 and compared the results with the initial values obtained in the case of the physical-design subset of Madeo toolkit. The “Overview Pyramid” of the obtained results is presented in Figure 15, and we can see that 4 of the eight metrics (green boxes) are now close to the Smalltalk average figures, compared to the evaluation of the legacy code-base, presented in Figure 3, for which all metrics are either too high or too low compared to typical Smalltalk projects.

![Figure 15. “Overview Pyramid” of physical-design system created during this work. The green rectangles emphasize the values closer to the average, as opposed to the red/blue ones which emphasize value closer high/low thresholds](image)

The amount of the deviation from a typical Smalltalk projects is shown visually in Figure 16. This figure shows that most of the metrics are in between the typical Smalltalk project limits, however the coupling axis still remains well beyond the high threshold values, especially in the case of the coupling dispersion which changes almost triples from a relatively low value, in the Madeo case, to a value 150% larger than the average. This shows that the transformation-based metaphor multiplies the number of classes involved in implementing the optimization policies required for the physical-design problem. Since the coupling intensity remains virtually unchanged we can infer that this high level of collaboration is typical for our system. Moreover, even if the system is more tightly coupled the use of the model-to-model transformation metaphor for algorithm design ensures that these dependencies are functional (domain-specific) and not an artifact of bad software design. However, these results still remain well beyond the high threshold values obtained showing the need for further evaluations to better understand the cause of this high amount of coupling.

The low value in the case of the average hierarchy height metric is explained by the extraction of algorithm frameworks from the initial code-base (e.g. the routing framework) which add new hierarchies to the overall system. In this version of the prototype these new hierarchies have a relatively low height mainly since they do not yet implement mature frameworks. In other words these frameworks are not yet populated with too many different classes which assure the typical functional variation and consequently create deeper inheritance hierarchies.

The very large difference between the initial value in the case of the high level structuring metric comes from the drastic change in packaging policy adopted during the development of the prototype presented in this study. In the case of the Madeo code-base the packaging policy was driven by intrinsic separation of the domain concepts, which in the case of the physical design is composed of three clearly identified elements: the application, the architecture, and the optimization policies. However, during the developments presented in this work, the packaging policy was oriented by the logical structure of the toolkit with the main purpose of isolating at maximum the software artifacts so that during the evolution process the minimum of packages are changed for any added or modified functionality. Thus, the overall number of classes per package decreased drastically from an average of 34 to 5 which represents 58% of the average in a typical Smalltalk system (see Table 1).

At the end we want to emphasize that the evolution process presented in this study may not be possible in non-Smalltalk environments mainly due to the lack of reflectivity, openness and
flexibility. But at the same time, imposing a certain development methodology might be hard mainly due to the same reasons. At times during the experiences described in this section we found ourselves wanting to hack our way into simpler (or faster) solutions by instantiating new objects as needed, creating unnecessary dependencies or using shared state just because its possible.

8. CONCLUSION AND FUTURE WORK

The Madeo FPGA toolkit served as trustworthy physical design back-end for enabling our research group to innovate in the challenging field of EDA. As the targeted domain is itself subject to major evolution, the EDA must reflect these changes: technology is driving the software evolution. After 15 years of various project specific modifications, and numerous integration cycles our environment started to degrade, and at time functionality was lost. We identified that an outdated, ad-hoc prototype-driven development process was at the core of these degradations. The legacy, however, remains highly valuable, and its loss is unaffordable to our group. In conclusion, if taming the complexity of today IC design lies in our capacity to reuse the results of the last 50 years of high-end EDA research, a limiting factor comes from the semantic loss of outdated models over time, along with an increasing complexity of the EDA framework. This prohibits newcomers to reach high productivity as the learning curve is flat: simply addressing technical issues is not enough, and to cut off the entrance ticket to the EDA skills appears as critical.

This study presents a solution to this problem based on a mix of MDE methodology coupled with the recent innovations within the Smalltalk community. Practically our approach relies on the FAME executable meta-modeling framework the development of domain-specific models, and the decoupling of the optimization algorithms from these models using the “transformation” metaphor.

The approach presented in this study opens the physical-design toolbox towards creating an open environment for research targeting physical-design automation on multiple technological targets. Through the adoption of a model-driven engineering methodology this toolkit prototype sketches the roadmap to a common way of modeling all structural aspects of the physical-design
problem at multiple abstraction levels, and provides the building-blocks needed for interoperability between development tools used for the specification, the design, and the verification of integrated circuits. However, this prototype doesn’t provide a complete physical-design environment needed for industry-level applications. To reach such a goal further research is needed especially on formalizing the models, and meta-models presented to offer a stable and expressive vocabulary for specifying all aspects of the physical-design process.

The “transformation metaphor”, used for algorithm design and specification, provides the infrastructure for creating highly modular optimization routines. Moreover, it doesn’t constraint the algorithm designer to use any particular programming language or formalism, as long as the resulting module provides clean and well-defined interfaces. This opens the way for future integration of hardware accelerated routines into the tool-flow, and along with the tool-flow reification can provide the infrastructure for side-by-side execution of hardware and software tasks to speed-up the physical-design process.

However, this process must be quantified in terms of expectations. This comes along with defining metrics. Rather than trying to isolate best metrics, we relied on a set of metrics against which to score the code refactoring and code redesign. This approach promotes a custom user-oriented energy function. Applying this methodology to our legacy code-base not only improves the software architecture of our solution, but it also opens the toolkit enabling plug-and-play algorithm reuse. A direct benefit was to smoothly schedule the software integration, protected from regression. Relying on these developments we were also able to easily prototype and test different automated solutions targeting new nanoscale architectures. In the future we plan to push ahead and formalize these concepts into a concrete Smalltalk based transformation engine, which will be able to provide the users with an environment for algorithm design and integration into domain specific tool-flows.

REFERENCES


