A Model-Based Approach to Reconfigurable Computing

By Daniel Taylor

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of
Master of Science
In
Computer Science

Shawn A. Bohner, Chair
Denis Gracanin
James D. Arthur

5 December 2008
Blacksburg, VA

Keywords: Model-based engineering, reconfigurable computing, FPGA design, hardware abstractions

A Model-Based Approach to Reconfigurable Computing

Daniel Taylor

(ABSTRACT)

Throughout the history of software development, advances have been made that improve the ability of developers to create systems by enabling them to work closer to their application domain. These advances have given programmers higher level abstractions with which to reason about problems. A separation of concerns between logic and implementation allows for reuse of components, portability between implementation platforms, and higher productivity.

Parallels can be drawn between the challenges that the field of reconfigurable computing (RC) is facing today and what the field of software engineering has gone through in the past. Most RC work is done in low level hardware description languages (HDLs) at the circuit level. A large productivity gap exists between the ability of RC developers and the potential of the technology. The small number of RC experts is not enough to meet the demands for RC applications.

Model-based engineering principles provide a way to reason about RC devices at a higher level, allowing for greater productivity, reuse, and portability. Higher level abstractions allow developers to deal with larger and more complex systems. A modeling environment has been developed to aid users in creating models, storing, reusing and generating hardware implementation code for their system. This environment serves as a starting point to apply model-based techniques to the field of RC to tighten the productivity gap. Future work can build on this model-based framework to take advantage of the unique features of reconfigurable devices, optimize their performance, and further open the field to a wider audience.

Acknowledgements

To my friends, colleagues, and everyone I have met at Virginia Tech, I would like to extend my sincere appreciation. You have made my experience in these few short years one to last a lifetime. I would also like to thank my family, to whom I owe a great deal of my accomplishments for their constant support. I would especially like to show my appreciation to my parents, who have always supported and encouraged me in everything I do.

I would like to express my gratitude to my research advisor Dr. Shawn Bohner for his guidance and support throughout my research experience. He spurred my interest in software engineering, inspired me to complete this work, and has guided me along the way. I would also like to thank Dr. Denis Gracanin and Dr. James Arthur for their encouragement and support on my committee. In addition, I would like to thank the members of the Center for High Performance Reconfigurable Computing at Virginia Tech from the Electrical and Computer Engineering department, Dr. Peter Athanas, Jorge Suris, and Adolfo Recio. Without their expertise, this research would not have been possible. The other members of the CHREC team, Dong Kwan Kim and Shyam Visamsetty, provided valuable assistance with their participation in this research.

Table of Contents

Acronyms

ADL	Architecture Description Language
ASIC	Application Specific Integrated Circuit
ASM	Architecture Specific Model
CHREC	Center for High-Performance REconfigurable Computing
CIM	Computationally Independent Model
EMF	Eclipse Modeling Framework
FPGA	Field Programmable Gate Array
GMF	Graphical Modeling Framework
HDL	Hardware Description Language
IDE	Integrated Development Environment
IP	Intellectual Property
JET	Java Emitter Templates
MBE	Model-based Engineering
MDA	Model-driven Architecture
MDD	Model-Driven Development
OCL	Object Constraint Language
PIM	Platform Independent Model
PSM	Platform Specific Model
RC	Reconfigurable Computing
SDR	Software-Defined Radio
XADL	XML Architecture Description Language

List of Figures

List of Tables

Chapter 1
Introduction

Software complexity is a challenge that software and reconfigurable computing engineers will continue to face as demand for software and reconfigurable devices continues to grow. There is an ever-increasing need for larger systems with higher complexity and more functionality than ever before. Customers demand that systems are secure, reliable, easy to maintain, and easy to use. System requirements for robustness, power usage, portability, and performance only continue to increase.

The technology and power to meet these needs is available, but there is a large and ever-increasing gap between the technology and the ability of developers to effectively use it. High performance applications like video processing, software-defined radio, digital signal processing, and many others require this gap to be closed if costs and development times are to be kept low. These demands cannot be met with traditional software implementations because of the performance limitations of microprocessors. Implementing these applications directly in hardware provides an opportunity to make customizations for performance

Hardware technology continues to advance at a rapid pace, but the programs that software engineers develop do not rise to meet it. This gap is especially visible in the area of reconfigurable computing, where the technology exists for very powerful, high performance computing applications through the use of devices like FPGAs, but programmability, usability, and the ability to reuse and maintain systems are very difficult to achieve without very skilled computer engineers and specialists [1][2]. Following Moore's law, the number of transistors available per ASIC is projected to increase, but the number that can be effectively used does not grow at the same rate [3]. The design gap between potential and what is actually used is holding back productivity. This hardware design problem can be extended to FPGAs, and may also be even more prominent due to the complexity added by their reconfigurability.

Software development was previously at a point where the increasing capability of devices was outstripping the ability of developers to create applications. The languages software engineers had did not allow them to easily express domain level ideas, similar to the situation currently found in hardware development. Software developers worked with registers and bits instead of dealing directly with the problem at hand. Hardware development is similar, in that engineers work with low-level circuit design, yet has several differences, because of the inherent properties of using a fully customizable architecture instead of a general purpose instructions.

Hardware device capacity continues to increase following the trend of Moore's law. While hardware design productivity has been increasing, it will never catch up to the capability of the devices at the current rate. The two trends of transistor availability and transistor usage will continue to diverge unless a significant advancement is made in productivity. In order for the RC market to thrive, the design productivity issue must be addressed [4]. Commercial success is unlikely while the knowledge barrier to programming reconfigurable hardware devices exists.

FPGAs have the potential to dominate the market if engineers can harness their speed and flexibility. Unfortunately, due to the design complexity involved with FPGAs, businesses may choose other solutions [4]. High design costs, long time-to-market, and difficult maintenance, all caused largely by complexity, may discourage use of FPGAs. ASICs will always be able to implement the same solutions as FPGAs, minus reconfiguration, at least as cheap, fast, and low power [4]. General purpose platforms trade hardware performance for programmer performance, providing the best in design time, flexibility, and maintenance [1]. FPGAs offer a compromise, taking a little from the best of both worlds, but only if developers can deal with the complexity necessary to harness that power.

The markets do not wait for developers to figure out how to use a technology and make it cheap and available. Other technologies will continue to advance and will gladly take the place in the market that FPGAs could fill. The RC community must find a way to overcome these challenges to make RC devices a commercial success [4]. FPGA technology will also increase, as mentioned earlier, but unless developers can actually make use of that potential and bridge the productivity gap, the field will stall.

One step towards solving the problems with FPGA development is to make the process of programming a reconfigurable device easier. Higher level language abstractions have already proven their ability to increase productivity for software. If hardware development can benefit in the same way that software did, then the gap between capacity and capability will begin to converge.

This research explores how RC can benefit from raised abstractions, particularly by applying model-based engineering principles. The approach, covered in Chapter 3, involves creating a modeling environment where developers build hardware applications with models instead of writing low-level code. The environment allows engineers to engage the development process at a higher level of abstraction, closer to the domain areas of their applications, enhancing their ability to develop systems. The purpose of this research is to open up the field of RC to using a model-based approach in order to build more complex systems faster and easier.

1.1 Overview of Reconfigurable Computing

One way that researchers have attempted to deal with some of the increasingly high requirements of performance in some software applications (data encryption [5], digital signal and image processing [6], and other high performance problems) is to use reconfigurable computing devices, such as a field programmable gate array (FPGA). RC devices provide engineers with the ability to partition programs such that part is run on software and part is run on hardware. For example, a hybrid computer might contain a general purpose processor that acts as a controller and specialized FPGA hardware that takes on computationally intensive tasks, like graphics processing, matrix calculations, or physics computations. Hybrid computers can take advantage of this partitioning, allowing heavy work to be done in parallel on hardware to greatly increase performance. FPGAs are flexible, relatively low cost and low power for the amount of performance they offer, but the complexity of the hardware and complexity of the applications are continuously increasing [1]. Although FPGAs offer great boosts in performance, they suffer in programmability [7]. Since the programs run as hardware, very low level

concepts like circuit design are involved in the development process, which decreases the ability of current software developers to work with the technology.

The advantage of being reconfigurable is what sets a FPGA apart from an application-specific integrated circuit (ASIC). An ASIC is a very high performance, low power hardware implementation manufactured solely for very specific applications. Once the ASIC has been created, it cannot be changed, and a new one must be produced to do something different. A FPGA‘s hardware can be reprogrammed in the field to carry out a slightly modified, or even a completely different task.

The fact that FPGAs are not based on the von-Neumann architecture separates them from a traditional microprocessor [1]. Because operations are implemented in hardware instead of software, they can be carried out much faster. For example, in one case study, a point multiplication implemented on a FPGA running at 66 MHz was 540 times faster than the software version running on a dual-Xeon computer running at 2.6 GHz [6]. Hardware implementations can do more in a smaller area on a chip than a processor, effectively giving them better energy consumption [8]. The benefits of parallelism, performance, and flexibility that come from the difference in architecture are very promising.

In order to decrease the design productivity gap, programming with the technology must be made accessible to engineers who are not experts in the field of RC. Current development environments are hard to use and require an in-depth understanding of the hardware systems, so relative productivity suffers and cannot match the growth in FPGA circuit logic. This means that much of the advantages gained by increased availability of circuit real estate are not exploited by the potential applications that could be developed for them. Further, as the situation persists, the investments in continuing the FPGA improvements will naturally diminish as they do not meet RC market expectations.

Development environments that aid developers in RC design are essential for increased use of these devices in systems [9]. Most environments today are geared towards FPGA experts or the environments are bound to one particular vendor or architecture. Software developers still have trouble reusing application level components to save time [10]. Without useful tools to support high-level development for RC,

programmers will be stuck in the low-level code, unable to make applications that meet the long list of demands desired by consumers. The productivity gap will continue to exist as long as developers are not given the tools and knowledge to fully utilize the technology. One way to arrest this productivity gap is to employ more application stakeholders in the development of RC applications. That is, if some of the programming could be accomplished by the application domain experts, then that would free up engineers to focus on producing the components and tools that these domain experts use to produce the relevant systems.

Consider the case of a software engineer who is an expert in building systems for a particular domain. Imagine the amount of wasted time if this developer had to create a system using only assembly code, or even C or Java without any standard libraries to build on. The expert's domain knowledge cannot be efficiently used because so much groundwork needs to be done, increasing the time-to-market for the application. Unfortunately, this is effectively what is being asked of RC developers. Customers want applications tailored to their domain, but the RC programmer has to work with circuit design, far below the domain area, to accomplish that goal. If an application domain expert had the tools necessary to build an application that leverages the work done by RC experts, then they could make effective use of their abilities and domain knowledge.

With the current set of tools, an expert RC developer is required for developing each and every kind of program by hand or knowing the intricate details of how to use the complex generation tools that are available. Other engineers do not have the background knowledge to work with architectures other than the von-Neumann standard [11]. Creating each component from scratch is very time consuming and takes away time that could be spent ensuring that an application meets requirements, testing, or optimizing. Unfortunately, the load cannot be shared, because there is a relatively small number of RC experts to tackle the vast array of reconfigurable hardware problems compared to the number of software developers there are to tackle software problems. An experienced craftsman does not want to spend his time working on basic building blocks that apprentices could build. An expert RC engineer's time is also important in the same way, so it would be beneficial if other developers could be leveraged on a

project without requiring extensive working knowledge of the low-level hardware aspects involved.

1.2 Overview of Model-based Engineering

One approach to opening up RC development to people who are not RC engineering specialists is by employing model-based development techniques. Model-based engineering (MBE) offers a way to deal with software complexity by raising the abstraction level at which engineers solve problems. By using MBE to design software, a developer creates a separation of concerns between the code and the application logic through models [12]. Abstract models are built to represent the domain, which are used to create more concrete models to solve the problem. Each model is elaborated and refined with further models until an implementation is reached. This design methodology offers benefits in reusability and portability due to the higher level of abstraction. Lower level artifacts are automatically generated, saving programmers' time and increasing productivity. The models also serve as documentation that can be used to help understand the system.

Round-trip engineering, with respect to MBE, is defined as synchronizing the different levels' models so they are always up-to-date and reflect changes in other levels throughout the design process [13]. Ideal round-trip engineering involves automatically updating lower level models to reflect changes in higher level models, and vice versa. Transformations between models are used heavily to achieve such automation. In Model Driven Architecture (MDA) [12], round-trip engineering is hoisted up as a key means for supporting and sustaining engineering and maintenance. The difficulty of automatic model synchronization in both directions is one major limitation of round-trip engineering, keeping it from being successfully employed in MDA. Detailed model transformations are necessary to link up the interrelated models without manual interaction. The opportunity for roundtrip engineering offered by the models and generation can provide traceability throughout the system, from higher level requirements and models down to code. Traceability allows developers to track requirements down to code for verification and also to trace code back up to see what effects certain changes

might have on the requirements of the system as a whole. Even partial round-trip engineering helps to support traceability, enabling better verification of requirements for a system.

Model-based engineering can be loosely defined as using models to describe artifacts from the software development life cycle. More specifically, models are created ranging from very abstract computationally independent representations of the domain all the way down to a platform specific implementation and code. These models form a hierarchy, allowing a developer to move from high-level abstractions down to low-level implementation through a series of systematic model transformations. At each step along the way, the system is refined and decisions are made that affect the levels below. Model-driven development (MDD) is the general process of using MBE principles to primarily create models that represent the system on several different levels, instead of code, to maximize productivity in the design process [14]. The Object Management Group adopted the MDA framework [12] in an attempt to standardize the way models are used in MBE for software development. The underlying idea of MDA is the separation of the specification of the system from the actual operation and implementation of the system. MDA provides three different viewpoints for reasoning about and modeling a system: the computationally independent model (CIM), the platform independent model (PIM), and the platform specific model (PSM). The different abstraction levels represent localized areas of change, where changing the system on one level will not have a large effect on the rest of the system.

There are several advantages to using higher level abstractions when modeling a system. Designing the system at a level closer to the application domain makes it easier to catch design errors early and ensure that the correct constraints and functionality described in the specification are expressed in the system [15]. Validations specific to each abstraction layer can be tailored to suit the needs of the application. High level validations ensure the overall correctness of the system, while lower level validations closer to the hardware might be included for specific platform optimizations or constraints. Reuse of lower level components is encouraged by working at a higher abstraction level, encouraging modularity and increasing productivity. The reduced

coupling between the application logic and implementation make the system easier to port between different implementation platforms.

The benefits that come from a higher abstraction level by using MBE for regular software development also apply to RC development and can increase productivity and reuse. Model-based principles allow separation of the application from the hardware by splitting the system into different models at different levels of abstraction. The split between the application logic and implementation platform is particularly important for RC, since what is necessary to implement a system in hardware on one platform may be very different for another platform. Being able to reuse components across platforms is one key advantage brought about by using model-based techniques. By providing a tool that can store high-level components in a repository for later reuse and a modeling environment that does not require expert-level knowledge of FPGA design, software engineers can save time by focusing on the application design instead of the implementation details.

In the next section, RC and MBE are examined together to motivate the problem, scope it to what is addressed in this research, and introduce the basic research approach. A recursive metamodel and model transformations are key features in this research taken from MBE to help solve some of the problems facing RC. A hardware application can be conceptualized, specified with the metamodel in the PIM, and generated to a PSM through model transformations.

1.3 Research Goals and Approach

The problems facing the field of RC have a strong parallel with the history of software development. Originally, computers were programmed at the microcode level and engineers worked with machine instructions. Assembly language provided a higher level abstraction for programmers to work with, which was transformed by an assembler to machine instructions. Eventually, higher level domain-specific programming languages like FORTRAN and COBOL came around. These languages were supported by a compiler that converted the source code into lower level instructions. As higher level languages and higher level abstractions were created, programmers could work at a

level closer to the domain of the application they were trying to create. New languages allowed programmers to do more with less code [16]. At each level, programmers gained productivity and the ability to reuse components at the level below, but they gave up optimizations on performance and some flexibility.

RC is still at the microcode stage, requiring programmers to have knowledge about very low-level circuit design in order to create an application. Various C-based languages exist that map to HDLs; however, those languages still need the programmer to think on the scale of circuits. As seen in software, major productivity gains will not occur unless the level of reasoning about problems changes. A program implemented on a circuit that takes advantage of the parallelism offered by hardware is much more difficult to create than a sequence of instructions to run on a processor [8]. In order to make gains in productivity, the abstraction level at which engineers create RC programs needs to be elevated. Along the way, some optimizations may be given up, but that will allow for the development of larger and more complex systems that could not be feasibly created otherwise. Several attempts are being made at doing this through developing higher level languages for programming hardware in a C-like manner [6][7][9]. Many of these approaches present a small step forward in raising the level of abstraction, but still require significant knowledge of RC programming to create systems.

The specific problem that this research addresses is directed at RC to show that technologies successfully used in software engineering apply to RC with some of the same types of results. Model-based techniques have been helpful in software engineering for designing larger scale systems and reducing the complexity that a developer works with. An integrated development environment (IDE) that supports MDD for hardware design can greatly enhance the RC development experience by enabling modeling abstractions, transformations between models, and integration of components. This in turn addresses the productivity problem by increasing the ability of developers to work with problems closer to the application domain. Higher level abstractions from MDD make it possible to involve more non-engineering resources, reuse work products, and support evolution and maintenance of systems while not being tied to any particular computing technology [14]. All of these advantages work together to help tackle the productivity gap in hardware by following the example found in software.

The approach presented in this thesis to solve the productivity problem is creating an Eclipse-based modeling environment where MBE principles can be applied to RC design. This research is one of the research initiatives at Virginia Tech's Center for High Performance Reconfigurable Computing (CHREC). Users will create a high-level PIM to represent the system, which is used to generate the application for a specific architecture based on an architecture specific model (ASM). The ASM helps bridge the gap between the PIM and the implementation. With the PIM and ASM, the PSM can be generated. The PSM will vary depending on what platform it is generated for. In the case of a FPGA, it would be the implementation HDL tooled for the architecture of the chosen FPGA. The intermediate models are represented in XML using xADL (XML Architecture Description Language).

Initially, the modeling environment will focus on composition, pulling together pre-made components to build a system, as opposed to transformation, which is the process of mapping the high level models to more implementation specific models and eventually directly to code. As developers create new components, they are added to a repository which stores them and makes them available for use in other applications.

The hypothesis of this research is that the benefits that MBE offers for regular software development can be effectively applied to the field of RC. By raising the abstraction level for RC development and separating the application from implementation, productivity can be increased by taking advantage of reuse and automation. Providing a unified development environment that is not tied to specific hardware architectures will enable greater portability. The main questions this research addresses are:

- Is using MBE for FPGA development feasible?
- Can reuse and portability be supported?
- Are productivity gains possible by using a model-based development environment?
- How do these compare with similar gains/losses in software engineering?

1.4 Thesis Outline

This thesis contains 5 chapters including the introduction to RC and MBE, motivation, and research objectives in Chapter 1. Chapter 2 provides background research on Model-Driven Architecture and Reconfigurable Computing. It includes a review of related research in the field of RC. The approach used in this research to model RC applications will be presented in Chapter 3. It includes a description of the metamodel used, how the modeling environment was built, the specific aspects of MBE that were applied, and a discussion of the demonstration carried out in the environment to model a transmitter for use with a FPGA.

Chapter 4 covers results of the demonstration and provides analysis of the approach from Chapter 3. It also relates the approach back to the research questions in the previous section. Lastly, Chapter 5 presents conclusions and a summary of the completed research.

Chapter 2
Literature Review

The trend throughout the history of software development has been to create higher level abstractions with which to interact with the computer. One of the more recent advances in ways to raise the abstraction level even higher is model-based engineering. MBE has been put to use successfully for some software applications in both industry and research and shows promise for being used in other fields.

Reconfigurable computing is a hot topic in research and industry, thanks in part to high-performance, reconfigurable devices like FPGAs. The potential for RC devices to increase performance for a wide variety of applications is very promising, as is their flexibility that comes from reconfigurability and advances in run-time reconfiguration [5]. RC technology has the potential to make a large impact on the way computing is done.

As long as RC devices are difficult to use, harnessing their potential will be a slow process. A number of published articles and research papers agree that a gap exists between the capability of the technology and the capability of the engineers to use it. Several different approaches are being taken to increase productivity; all are focused on creating abstractions that reduce the burden on the developer.

2.1 Model-Driven Architecture

Model-based engineering is essentially the process of using models to describe artifacts throughout the whole software development life cycle. MDA [12] is a framework developed to apply MBE to software engineering. The underlying idea of MDA is the separation of the specification of the system from the actual operation and implementation of the system. The specification describes what the software system

does, and the operation and implementation describe exactly how the system accomplishes the goals set forth within the specification.

The separation of concerns allows for improved portability and reusability, allowing the system to be divided up into reusable components. MDA provides three different viewpoints for reasoning and modeling about a system: computationally independent, platform independent, and platform specific [12]. Models are well-defined combinations of text and diagrams that are created at these different levels of abstraction and can be transformed into an implementation through mappings and transforms [10]. Code is also a model, one that can be directly executed by a computer.

MDA uses UML, one of the most widely used modeling languages, to create the necessary models for the different levels of abstraction. By combining UML with Object Constraint Language (OCL), precise and powerful models can be created [17]. UML is popular because different subsets and profiles of UML can be used, like Executable UML, which can be easily mapped to an implementation. In MBE, any representation of a model can really be used, as long as one is willing to write the necessary transformations to map it to the next level of models.

MDA breaks down the design into three different levels of abstraction, the computationally independent model (CIM), the platform independent model (PIM), and the platform specific model (PSM) [12]. The CIM represents the domain level knowledge about the system, independent of how the problem gets solved. The PIM shows the application logic, but not any of the implementation details. The PSM explains how the system is implemented, and is the level that executable code is generated from. Transformation between these models is one of the biggest advantages of using MDA [17].

Figure 2-1 provides a visual representation for how a CIM is transformed to a PSM [12]. The transformation specification includes all of the transformation rules necessary to map any of the allowed platform independent types to platform specific types. When a PIM is transformed, it simply uses those rules to create a new PSM based off of it that represents the exact same system. Since the PIM is completely independent from the platform, different platforms can be chosen, and all that needs to be done to create the PSM is to change the transformation rules.

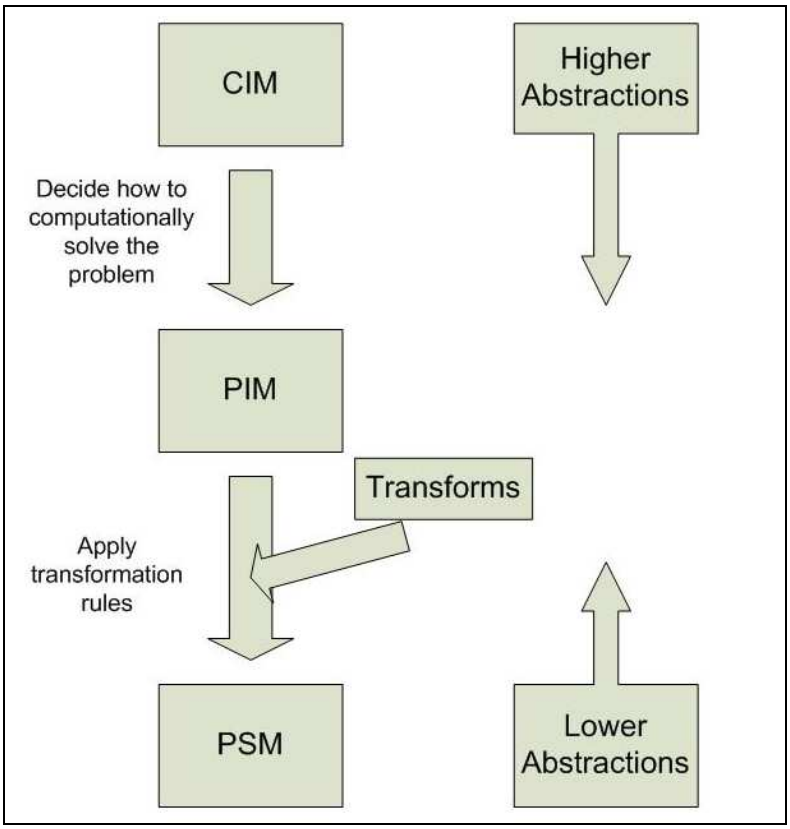

Figure 2-1: Model Transformation. How the platform specific model is generated from the platform independent model, which was derived from the computationally independent model in MDA [12].

Several case studies have been conducted to see whether MDA can deliver on some of the promises it makes. In one case study [18], a large commercial bank took on an Enterprise Application Integration (EAI) project, with goals of transaction integrity, performance, high availability, flexibility, and consistent design. They did not want to use a traditional approach to engineering this system because of the large number of systems/platforms involved. The programmers decided to use Borland's Together MDA technology to support a model-based development approach.

They worked from higher to lower level abstractions, first using UML 2.0 to develop the PIM, which contained the structural relationships of domain objects. Next, they transformed the PIM to a Java PSM. During the transformation process, they had to make several design decisions, like selecting collection types, using Currency objects, and using the Java Date object. These changes were manually implemented into the automatically generated PSM. A separate data model was also created in the PSM using the Query / View / Transform MDA standard based on available sample projects. Once

the PSM was complete, they generated the implementation details, such as code, configuration files, Data Definition Language and SQL scripts, and XSL through various transforms.

In this case study [18], the bank created artifacts at each level of abstraction from high level requirements down to a Java code implementation. The process started off with requirements, which were used to develop a PIM in UML 2.0. This represented the system without any of the implementation details (language, data model, etc.). The PSM was generated from the PIM to get a UML representation of the system, along with a data model. This set of UML diagrams was specific enough that it could be used to generate the system architecture for the Java platform. Once several design decisions were made manually at that level, the Java code was generated.

This software development project was a success, and allowed the bank to use it as a proof of concept for MDA development in the future where they might use it to create new adapters or wrap existing or legacy applications. By using MDA, they were able to benefit by decoupling the domain from the implementation, reduce expenses by using code generation, generate formal documentation in the form of models that actually represent the system, and gain a quicker understanding of the application [18]. This study shows that using MDA is a viable approach to software development in a real world setting.

Another case study [19] examined the viability of different modeling representations and if they could successfully be transformed to the lower abstraction levels. The study developed two PIMs, one with UML and one with EDOC (Enterprise Distributed Object Computing). Both were transformed to three different targets using the ATL (Atlas Transformation Language): Java, Web Services, and JWSDP (Java Web Service Developer Pack). EDOC provided a better representation than UML because it was more specific to the area, whereas UML is a very broad modeling language. They noted that the success of an MDA project relies on the quality of the meta-models and the quality of the transformations between models. The paper also noted that a previous study had shown that using an MDA tool made development 35% faster than using the standard approach [19].

A case study on using MDA for complex middleware solutions [20] found that MDA was great for doing rapid prototypes, but solutions developed with it lacked in the area of performance. The most important thing in a MDA project is the transformation process, and a language that supports the modeling and transformation will greatly improve the development process. The study noted two main types of transformations, model to model transformations within an abstraction layer, and transformations to different models across abstraction layers. The consistency in these transformations is a key factor in a successful MDA project [20].

The Cougaar agent-based architecture [21] is a framework for building large scale agent-based systems. An agent is a collaborative software object that makes decisions based on the environment [22]. In Cougaar, multiple agents work together to carry out some tasks [2]. A model-based approach using MDA was carried out by Virginia Tech [2][22] in order to work with this framework. The researchers developed an IDE prototype that used a model-based approach to work with Cougaar and develop applications using it. The Cougaar MDA approach made use of workflow diagrams at the CIM, a generic domain application model for the PIM, and a generic Cougaar application model for the PSM. The CIM detailed the requirements of the system and the PIM detailed its workings. The user builds components in the PIM based off elements in the PSM. Canonical components were identified in the example projects that could be singled out for reuse and added to a common repository. By providing a modeling interface and readily available components, a novice user could make significant productivity gains [2]. The research described in this thesis is an extension based off what was learned from the Cougaar project about applying model-based engineering to a domain.

Hardware developers already use several techniques from MBE in many different projects, although they may not call it that or realize it falls under the same category. Computer-assisted design tools are a form of model-based engineering. Tools like the Xilinx Core Generator [23] or LabVIEW [24] give users low-level abstractions to work with hardware. These and other tools provide graphical modeling environments to users with varying degrees of abstractions in their component repositories. Often, the repositories contain low-level circuit components that can be wired together to represent

an actual circuit. The kinds of abstractions offered in the repository determine the degree of productivity gain the tool offers. If a tool only makes logic gates available, there is not much reason to use the tool. However, if higher-level abstractions provide larger components, then users can get some benefit out of the tool. The idea that abstractions need to be raised is a common theme in a hardware tools and research.

One of the main research focuses is hardware compilers, which offer some of the same benefits as MBE, like higher abstractions, portability, and automation, but to a lesser degree and without the additional benefits of models. Architecture independence is also seen as a key point for improving productivity. Research has been done on mapping platform independent algorithmic representations of applications onto reconfigurable hardware [26]. There are also many initiatives to improve reuse through abstractions and patterns, some of which are covered later in this chapter. MBE brings each of these research goals together and offers a way to combine and integrate them to further increase developer productivity. The principles behind MBE are present throughout hardware research, but no development environment has been produced that unites all of these goals into a single integrated tool for developers to use.

2.2 Motivation for Reconfigurable Computing

In the history of computing, the von-Neumann architecture has been a common way to design general purpose computers and processors. The separation of the central processor from the data and instructions provides a convenient method of programming many different functions into a single machine in a relatively simple to reason about format. Von-Neumann machines can be classified into two different platforms, general purpose processors (GPPs) and application specific instruction processors (ASIPs) [1]. A GPP is found in all kinds of computers to do general processing and carry out a wide variety of tasks. An ASIP is developed for a specific purpose, usually with instructions that match up with a specific task and can carry it out quickly and efficiently.

Devices that do not use the von-Neumann architecture can avoid the von-Neumann bottleneck [25]. While the architecture provides a nice way to think about programs, it suffers from a performance bottleneck that occurs when transferring data

between the processor and the storage mechanism. Machines that do not use the von-Neumann architecture can avoid this bottleneck and implement faster processes through parallelism. These machines can generally be classified as FPGAs or ASICs. FPGAs allow the user to define the architecture and reprogram it, while ASICs are essentially just circuits and are fully customizable, but not reprogrammable [1].

The traditional FPGA architecture consists of a logic fabric containing look-up tables, flip-flops, and additional components along with a routing mechanism [27]. Look-up tables with 4-6 inputs are typically used as the basis for the configurable logic blocks. These components can be configured for a high degree of parallelism, which is difficult to achieve with the von-Neumann architecture. The routing architecture of an FPGA is also very important, as it determines how the logic blocks are connected to each other and the distance that must be traveled to reach from one block to another [28]. The logic blocks can be configured to specific types of functions and then programmed to be used in parallel [29].

Figure 2-2 gives a comparison of some of the important qualities of ASICs, FPGAs, and the two von-Neumann based platforms: ASIPs and GPPs [1]. From the figure, it is evident that the devices based on the von-Neumann architecture provide less performance and higher power consumption than ASICs or FPGAs, but they are very flexible because of their general application area. ASICs are very fast and low power, but completely inflexible and difficult to make. FPGAs can harness some of the power of user-defined architecture while still being somewhat flexible thanks to the fact that they can be reprogrammed [1]. However, FPGAs will not catch on unless they can lower their time to market and time to change code functionality by breaking away from their hardware oriented programming nature [4].

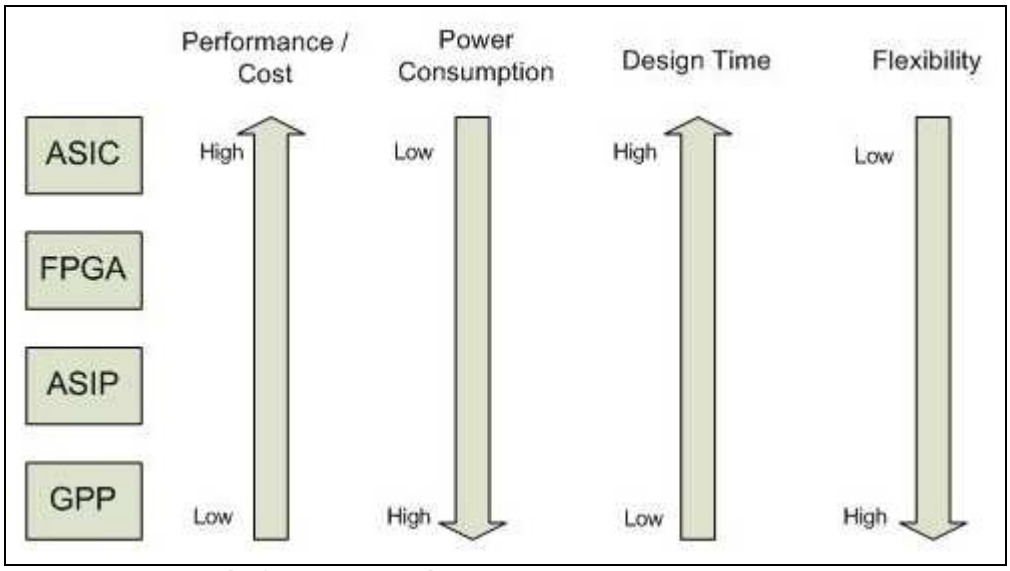

Figure 2-2: Computer Platform Comparison. A diagram showing the general properties of the four different computer platforms [1]. The bottom two are based on the von-Neumann architecture.

Reconfigurable development currently suffers in the area of programmability. In order to take advantage of some of the features that a RC device like a FPGA can offer, the application must be created at the hardware level. Many processes are thought of as a sequence of related steps that will be completed in order. Most software programming is done in a procedural manner as well. Although this has changed some with the advent of object oriented programming, a programmer still lays out their instructions sequentially within an object's methods. To take full advantage of the hardware benefits of RC, the application has to be converted to a parallel circuit that is often significantly more complex than the sequential way of accomplishing the task [8].

Taking advantage of the freedom afforded by not using the von-Neumann architecture has several advantages. Depending on the application and device, an FPGA can achieve orders of magnitude better performance than a traditional software implementation [8]. By utilizing a higher degree of speed and power, the same amount of work can be accomplished faster, resulting in average energy savings of 35% to 70% [6]. The potential of RC is too high to ignore.

Even with the clear benefits of RC, reconfigurable devices have yet to see widespread use in many application areas. The gap between developing applications for software and developing applications on hardware is large. Many developers have no way of crossing this gap without the assistance of tools that will support their design

efforts. The rest of this chapter covers research relevant to bridging the gap and making the hardware development process more accessible to the average engineer.

2.3 Hardware Compilers

In software, compilers were introduced because of the need to be able to create applications quicker and easier. The higher level languages that were made possible by compilers allowed programmers to move past assembly code and start using languages closer to the domain of the applications they needed to build. These languages let programmers deal with more complexity and build larger applications than would have ever been possible with low level assembly code.

Given what compilers did for the software development community, it is easy to see why the creation of a hardware compiler is one of the most common approaches to bridging the productivity gap for reconfigurable design. The current standard in industry is to use hardware description languages like VHDL or Verilog [6]. These languages resemble software languages by having similar syntax and control structures, but introduce functionality specific to hardware and concurrency. Register transfer level descriptions of circuits are used if the user wants to be able to generate an actual circuit from their HDL. These descriptions specify a circuit's behavior with registers, logic components, and the flow of data between them.

In order to achieve significant productivity gains for hardware application design, programmers need to be able to move beyond HDLs and describing circuit components. Creating a full system from scratch is too time consuming and costly for developers. In addition to design time costs, one must also consider the amount of quality assurance and testing that needs to go into checking such low level designs [30]. Compilers offer a way to raise abstractions and increase reuse in order to raise productivity. For general purpose design, there are two main research approaches to compilers: annotation/constraint-driven and source-directed compilation [6].

The goal of the annotation and constraint-driven approach is to supplement programs written in a regular software programming language like C with annotations in the code and configuration files [6]. Table 2-1 shows some examples of this approach

and some of their features. The advantage of this method is that there are generally fewer required changes to a program in order to compile it to hardware. Since the language remains the same, the programmer only needs to learn to use the annotations and constraints correctly. The disadvantage is that if the developer were to restructure the application in a HDL, it would probably be significantly more optimized than what the compiler can do.

Table 2-1: Hardware Compilers

Compiler	Approach	Target Architecture	Notable Features
Streams-C [31]	Annotation / Contraint-driven	Xilinx FPGA	C to VHDL, coarse-grained parallelism, automatic low-level optimizations
Sea Cucumber [32]	Annotation / Contraint-driven	Xilinx FPGA	Java to hardware, library with multiple implementations
SPARK [33]	Annotation / Contraint-driven	LSI, Altera FPGAs	C to VHDL, scheduling, resource binding, and state machine generation
SPC [34]	Annotation / Contraint-driven	Xilinx FPGAs	Vectorization, loop transformation, retiming, memory optimization
ASC [35]	Source-directed Compilation	Xilinx FPGAs	C++ interface for low-level coding, object-oriented libraries
Handel-C [36]	Source-directed Compilation	Actel, Altera, Xilinx FPGAs	Extends C for hardware programming
Haydn-C [37]	Source-directed Compilation	Xilinx FPGAs	High level component-based design extension to Handel-C
Bach-C [38]	Source-directed Compilation	LSI FPGAs	Extends C, potential for increased parallel optimizations

* Summary data from [6]

When compilers were first introduced in software, one argument against them was that a good developer could write optimized assembly that did the job more efficiently than a compiler could. Software developers do not write programs in assembly today for a few reasons. Compilers have advanced to the point that they can carry out a very large

number of optimizations that the average programmer would have no idea how to do. Even if an expert assembly coder could make better optimizations, the compiler can make a reasonable amount of optimizations much faster on a much larger amount of code. Overall, the compiler eventually took over the low level worries of programmers, allowing them to make larger systems at a higher level of abstraction.

The goal of the source-directed compilation approach is to increase the capabilities of a programming language to include everything a developer would need to explicitly describe hardware interactions [6]. Table 2-1 shows some examples of this approach and some of their features. With this approach, the source programming language is altered to account for hardware programming. New types, operators, control structures, parallel mechanisms, and other constructs are made available. The new language features give programmers increased flexibility, but a program might need to be extensively restructured and rewritten to take advantage of the new features.

A variety of commercial tools are available to offer developers the capability to quickly generate reconfigurable components or parts of systems. Xilinx provides its own set of vendor tools, like the Xilinx ISE [39] and Xilinx Core Generator [23], to work with its own FPGA products. Altera [40] provides its own set of design software and intellectual property (IP) cores optimized for Altera boards. Other tools exist that translate C-like languages to HDL and optimize HDL to an actual circuit [6]. Such a great assortment of design tools gives a RC expert everything they need to develop a hardware application if they know exactly what they want and exactly what they need to fulfill their requirements.

The problem with such a huge assortment of design tools is that not everyone is an expert who knows how to use each tool and how to choose between them. Another problem is shown in Table 2-1 by examining the architectures that the compilers target. Because of the large number of tools, vendors, and architectures, there is no common design tool that can be used between all of them. A developer is in trouble if they're using one vendor tool for a specific architecture and then they run into a problem that requires them to change architectures to take advantage of some other tool's feature or IP cores. The number of tools is partly caused by market conditions, but the field of RC as a

whole suffers if developers are forced to stick with certain vendors and cannot enjoy the reuse and portability benefits that come from unified development environments.

Both the annotation and constraint-driven approach and the source-directed compilation approach can be very effective in allowing programmers to make use of higher level language constructs to develop hardware applications. However, these approaches are only increasing the abstraction level by a small amount. To get the level of productivity gain necessary to catch up to the technology gap, an even greater elevation in abstractions is necessary.

High-level languages are also being built to support partial reconfiguration [42]. FPGAs are known for their ability to be configured at any time, even while parts of them are being used. This dynamic is what sets RC devices apart from other hardware and relates them to the software paradigm, providing flexibility incomparable to other hardware devices. The aspect of programmability lets RC act like software and in turn begs for the same support software has for developing applications and reusing design patterns. Unfortunately, little has been done commercially to provide software and tool support for partial reconfiguration at run-time [42]. As techniques that have been successful for dealing with software problems are applied to the RC design process in the future, it will be possible to provide increased tool support for developers to take advantage of the utility of RC and its similarities with the software paradigm.

2.4 Hardware Design Patterns

Design patterns are used throughout all software development projects. A design pattern describes a recurring problem, the standard solution to the problem, and the results of using that pattern [41]. Without design patterns, a software developer would have to start from scratch every time they wanted to build a new application, which is completely infeasible. Patterns are critical to quickly evaluating possible solutions to common problems.

As hardware design continues to scale up in complexity, developers have been looking for solutions to increase their productivity. The idea of reusable IP cores is one way to get more productivity by reusing work that has been done on other projects [30].

However, a developer has to select a core, configure it to the current application, and then manually interface it with the rest of their design, all of which can take a large amount of time. Reusing individual cores is only the first step in increasing the amount of reuse that can be achieved in hardware.

In software, large scale reuse does not come about by reusing individual classes and objects. Instead, software patterns are applied to help design the architecture of a system, giving guidance based on experience on a larger scale than individual components. To get the most out of hardware design patterns, they should also represent system level design decisions [30]. Creating a set of formal design patterns will allow hardware designers to reuse best practices and save time in system design.

Design patterns alone are not enough to bridge the productivity gap. If an assembly programmer has a good set of assembly coding patterns available for reuse, they will be able to increase their productivity at writing good assembly. However, the programmer is still writing assembly and has not raised the level of abstraction of the model they are using. Ideally, those low level design patterns would go into a compiler that utilizes those best practices automatically, saving the programmer time. By using the compiler and working with a higher level language, the programmer can get more reuse out of their work and also begin developing higher level design patterns that will further increase their capabilities.

Hardware skeletons [1] are a proposed solution to bridge the productivity gap between technology and programmer capability by raising abstractions while keeping efficiency. A hardware skeleton is defined as a reusable framework that contains optimization rules and takes variables and functions as parameters to implement some functionality. A design is composed of a hierarchy of skeletons to achieve efficiency through abstractions. A wide variety of hardware skeletons have been combined in a library to support pipelining, parallelism, reductions, and other common hardware tasks.

The hardware skeleton approach is very similar to a model-based approach. The goal of both approaches is the same: raise the abstraction level for developers. The hardware skeleton approach achieves this goal by providing a library of skeletons that represent the structure of a hardware application. The developer then chooses and fills in the parameters of those skeletons with the desired application logic. What is missing is

the metamodel for the transforms and refinements that enable the generation of the application from the constituent components.

For a model-based approach, the library would contain a variety of different components for the user to connect together. The metamodel and transformation rules provide the requisite infrastructure to know where to get the information necessary to assemble or compile the elements of the model into an application. It would be up to the transformations between the model abstraction layers to determine the appropriate way to assemble those in hardware, opposed to a user-parameterized skeleton with its own optimization rules. Similar to the hardware skeleton approach MBE provides for the human-in-the-loop to provide specifics that may not be recorded or complete.

In a way, the hardware skeleton approach presents reusable hardware design patterns, allowing the user to fill in the details of each pattern (pipeline, reduction, etc.) with their desired functionality (addition, functions, etc.). The model-based approach attempts to encapsulate functionality in cohesive blocks for reuse. These two approaches might work extraordinarily well if used together. Consider the advantages of being able to select a specific operational skeleton model and then plugging specific functional models into it to achieve the end result application. At this point, the user has been completely separated from the hardware, and optimization rules can be carried out when generating to the executable code. If higher level skeletons are created, breaking away more from the hardware, then hardware application development will begin to reach closer to the domain of the intended applications.

Template-based methodologies [43] are another approach to hardware design that shares some similarities with how MBE deals with software. While hardware skeletons provide an overall hardware framework for a task in an application, templates let a developer instantiate a specific architectural logic block and explore the solution space through parameterization. The developer can then use that architecture to implement their solution. A template is defined as a parameterized model representing a logic core for a reconfigurable architecture. Specifying parameters creates an instance of the template that can be targeted towards a specific application domain. Each template has several different layers within it, from the basic logic element to a complex array of blocks that make up the logic core.

The purpose of using templates is to reduce the amount of time required to complete an initial design and allow for quick exploration of different possibilities within an architecture design space through the parameters [43]. Instead of having a library of components to choose from, the template allows the designer to specify the parameters and generate the logic core based on those parameters. Parameterization of components allows the designer to build many more variations of possible components than would be possible to reasonably put in a library.

The model-based approach in this thesis also heavily relies on parameterization of components to allow designers to have freedom in the types of components they build. One difference with the model-based approach is that at the platform independent level, none of the parameters necessarily have a direct meaning to the implementation. Instead, the parameters at that level all relate to the application's logic. During the model transformations to reach the implementation, the parameters are used to make decisions specific to the implementation platform. Additionally, the platform and architecture that the application is being implemented on can be thought of a parameter for the model-based approach.

2.5 Platform-Based Design

The overall goal of system development is to go from a high-level (abstract) set of requirements all the way down to a low-level (concrete) implementation. The analysis and design decisions that are made throughout this process determine the properties of the resultant system. The idea behind platform-based design [3][44][15] is to map a system specification onto a platform instance. A platform is an abstraction layer somewhere between the requirements and implementation that has a set of elements, which can be composed to represent the system. Platforms can be built on top of each other, abstracting away the details of the lower level design [45]. Platform-based design methods are typically used in ASIC design to achieve high levels of reuse and integration in hardware [3]. By using a set of verified components, the work of developing a system shifts from component development to component integration.

Platform-based designs are usually created for particular application domains [44]. By sticking to a domain, the platform can provide optimizations tuned specifically for that application area. To work with a platform, a developer first chooses what types of components they need to suit their application, which leads them to a specific platform instance [44]. The developer then maps their application onto that platform, giving them an estimate of performance that system implementation will have. A design is refined by continuing the process, stepping through multiple platforms to achieve a more optimal solution [15]. Reusing the platforms and their elements gives productivity gains and also provides modularity to the system design, since the components have their own defined functionality within the platform [3]. A system platform is usually made up of software and hardware modules set up to communicate a certain way [43].

The platform-based approach to ASIC design has several things in common with MBE. The MBE design process involves taking the application requirements through model transformations to end up at a final implementation. At each stage of the model, decisions are made which further refine the system towards a specific implementation. Domain specific modeling languages are also common for designing software systems using model-based design methods. This lets the developer of the modeling language create rules that will benefit the user when it comes to generating a lower level model based on that design.

A model-based approach to hardware design has the advantage of breaking away from the implementation architecture and starting with a very high level model. A platform is typically associated with a specific architecture or family of architectures. After choosing one platform to work with, the developer can work with all of the elements available in that platform's library, but it may be difficult to switch to another platform if the need arises. The model-based approach is platform independent, allowing the system logic, not just the architecture components, to be reusable and also portable. MBE is partially an extension of the platform-based design approach into a much higher level of abstractions.

2.6 Summary

Reconfigurable computing has a large potential to increase the performance and reduce costs for a wide variety of computing application domains. High-performance reconfigurable devices offer the flexibility of general purpose machines with performance closer to specialized circuits. The challenge is tapping that potential and making it widely available.

Many different approaches are attempting to solve the problem in different ways. Hardware compilers are moving the programming abstraction up one step to take advantage of what engineers know about programming in software languages. Hardware design patterns help developers solve common, well-understood problems in a standard way. Hardware skeletons and templates allow designers to reuse architectural patterns and components and customize them with parameters. Platforms provide reuse of established libraries of modules that can be optimized for specific architectures. Hardware developers are continuously searching for ways to improve the level of abstraction at which applications are built. Raising abstractions is the best way to make a large jump in productivity to catch up with the potential of the technology.

Model-based engineering has proven to be a useful approach in software development for raising abstractions and enhancing the capabilities of developers. It has been successfully applied in both industry and research to increase reuse and leverage automation for increased productivity. If a model-based approach can be successfully applied to reconfigurable development, developers should see a substantial increase in hardware design productivity.

Chapter 3
Model-Based Approach to Reconfigurable Design

Model Driven Reconfigurable Computing (MDRC) is an attempt to build upon MDA to offer support for designing RC applications. RC development is currently facing increased demand for applications, a growing gap between developer ability and technology capability, and demand for increasingly larger systems with higher complexity [2]. Model-based engineering has worked to help meet the needs of software systems through MDA and MDD. A model-based approach should be able to provide similar benefits for RC.

The differences between software and hardware make the process of applying MDA to a hardware system a non-trivial problem. The PIM and PSM do provide for separating the platform from the program logic, but there is also the consideration of the architecture of the chip and board being used. Additional concerns include the area/space constraints inherent in hardware, performance optimization, and timing. Hardware abstractions can work towards solving some of these problems, but finding a solution to the optimization problems requires detailed research into the transformations that take place going from the high level models all the way down to the hardware.

The key premise of MDRC is the representation of the hardware application with models. The models provide abstractions to let the developer build an application separate from the hardware architecture and platform that it will execute on. Models are transformed down to code, which is also a model, through a series of transformations involving specifications of the architecture and platform. Figure 3-1 shows the different abstraction layers of MDA and how MDRC fits into them. From the diagram, it is clear that the division between the PIM and PSM is not discretely defined. In fact, transitioning between the different layers of abstraction can be thought of as a continuous process. Decisions made at the higher levels can have an effect on how the lower levels are generated.

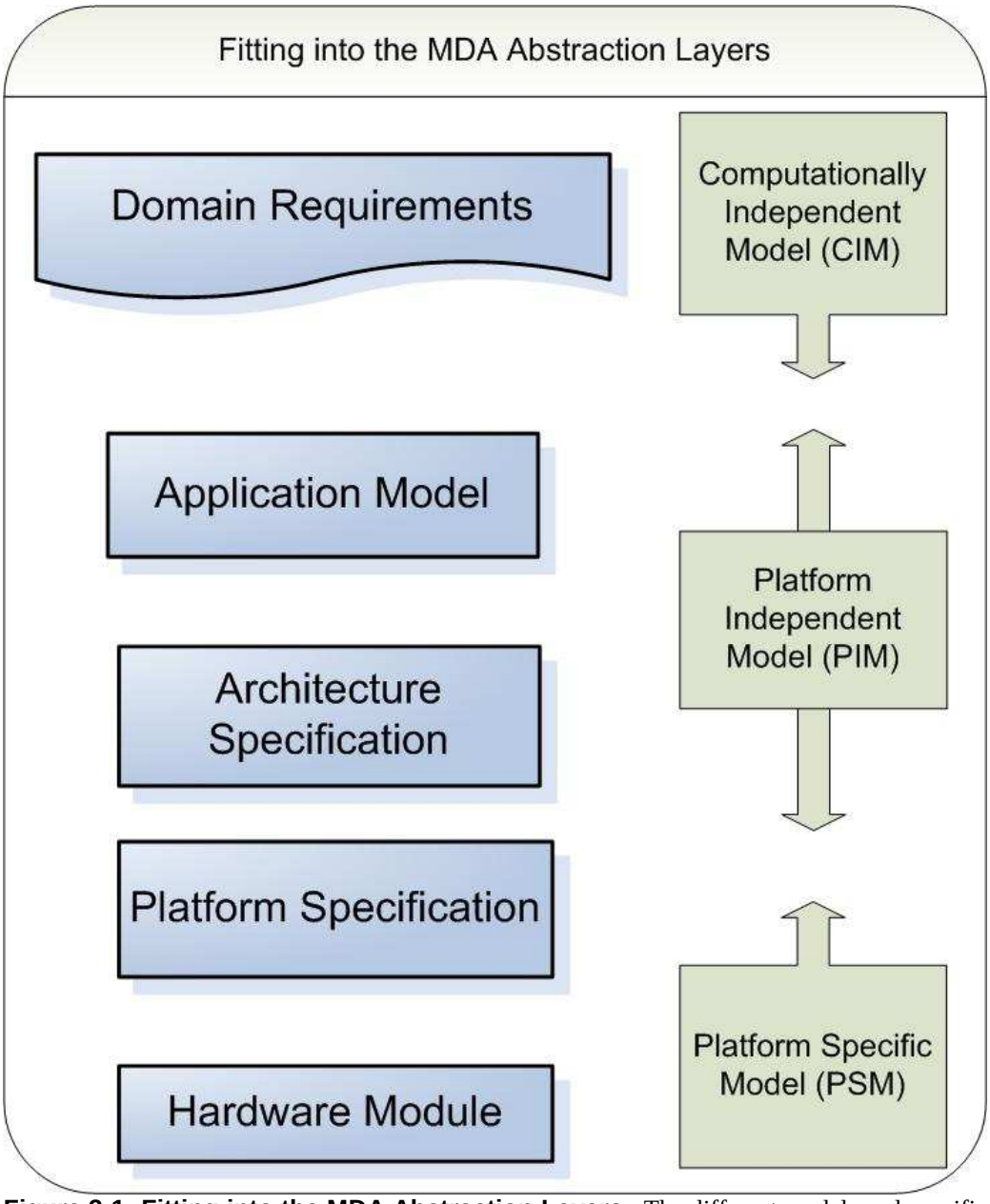

Figure 3-1: Fitting into the MDA Abstraction Layers. The different models and specifications used in this approach and which abstraction layer they are used in. Each abstraction level has several intermediate models that are used in the transformation process from abstract domain requirements to a platform implementation.

Similar to MDA, the CIM represents the domain that the user is trying to model, before any decisions are made on how to computationally carry out the domain objectives. The user models the application logic for the PIM within the Eclipse-based modeling environment. In this step, users assemble components that have already been developed and made available for use. Some of the components may be predefined for use with general applications, while others may have been created by other developers

and placed in the repository for reuse. Once the user has chosen, configured, and connected their components, the model is sent to a transformer that takes the PIM representation of the application, the architecture specification for the components, and the platform specification and generates the platform specific artifact, usually some type of HDL code.

One key feature of MDRC is how all of the models are described using XML. XML offers a highly extensible standardized way to represent the various models and specifications. In the PIM, the application model is based on xADL [46], a XML-based architecture description language. The xADL language provides canonical structures for modeling system architecture and the capability to specialize the language towards a particular domain. MDRC makes use of the generic xADL schema to represent the application logic with components, interfaces, connectors, and links.

3.1 Modeling Environment Architecture

One of the main goals of this research is to find a way to increase productivity in the area of RC development. The research explored how a model-based approach could enable a greater degree of reuse along with the ability for non-specialists to help out in the design process. The MDRC environment is one step in creating a more automated development process where RC engineers can team with other developers, collaborate, and make the most of work that has already been done. In software, many problems have already been encountered by others, and it is important to be able to make use of those accomplishments instead of trying to reinvent a new solution each time.

The ideal model-based environment would be completely model driven, and require no need for any developer to provide any low-level coding or modifications. According to MDA, automatic transformation best suits a mature component-based development setting where all of the possible architectural decisions below the PIM layer have been delineated and can be reused [12]. Reconfigurable computing is clearly not such a setting at this point in time, though it may have the potential to be so in the future. Taking this into consideration, this research has taken an assembly based approach as a

reasonable starting point for integrating MBE and RC, instead of a transformational approach.

The MDRC environment was developed in Java using Eclipse. It heavily employs the Eclipse IDE [47] and related plugins and frameworks. The modeling environment allows users to create high level models for a reconfigurable application, specify their parameters, and generate the low level code. The code is generated by combining the model specification with the architecture and platform specifications. The process then uses a makefile that selects the appropriate pre-existing code or leverages other tools to generate the necessary hardware modules with the correct parameters. The following are the main elements of the MDRC environment:

- Graphical Editor: the Eclipse Graphical Modeling Framework (GMF) [48] was used to create the basic graphical model editor. The editor is an interactive, visual representation of the application where a user can create and edit application models. The user can drag components into the model, configure their parameters, and drill down into the component hierarchy.
- Model Repository: the Subversive [49] Eclipse plugin provides basic version control and storage for models using Subversion. The repository provides a common place for models to be stored and accessed by others, enhancing collaboration and reuse.
- Model Compiler: a generation facility that utilizes JET to transform the graphical model into a XML format with specific hardware modules and their connections. The compiler then uses a makefile to combine the transformed model XML with the architecture and platform specifications to generate HDL. The makefile makes calls to several tools to carry out this job for a specific architecture and platform, detailed further in Section 3.4.
- Model Tree Viewer: a detailed view of all entities in a model. The tree view allows the user to view the whole hierarchy of components and edit their properties. It essentially allows the user to edit the underlying model directly. Figure 3-4 shows the tree editor view of the model. Components are listed in a hierarchical manner, allowing the user to drill down to the level they are interested in.

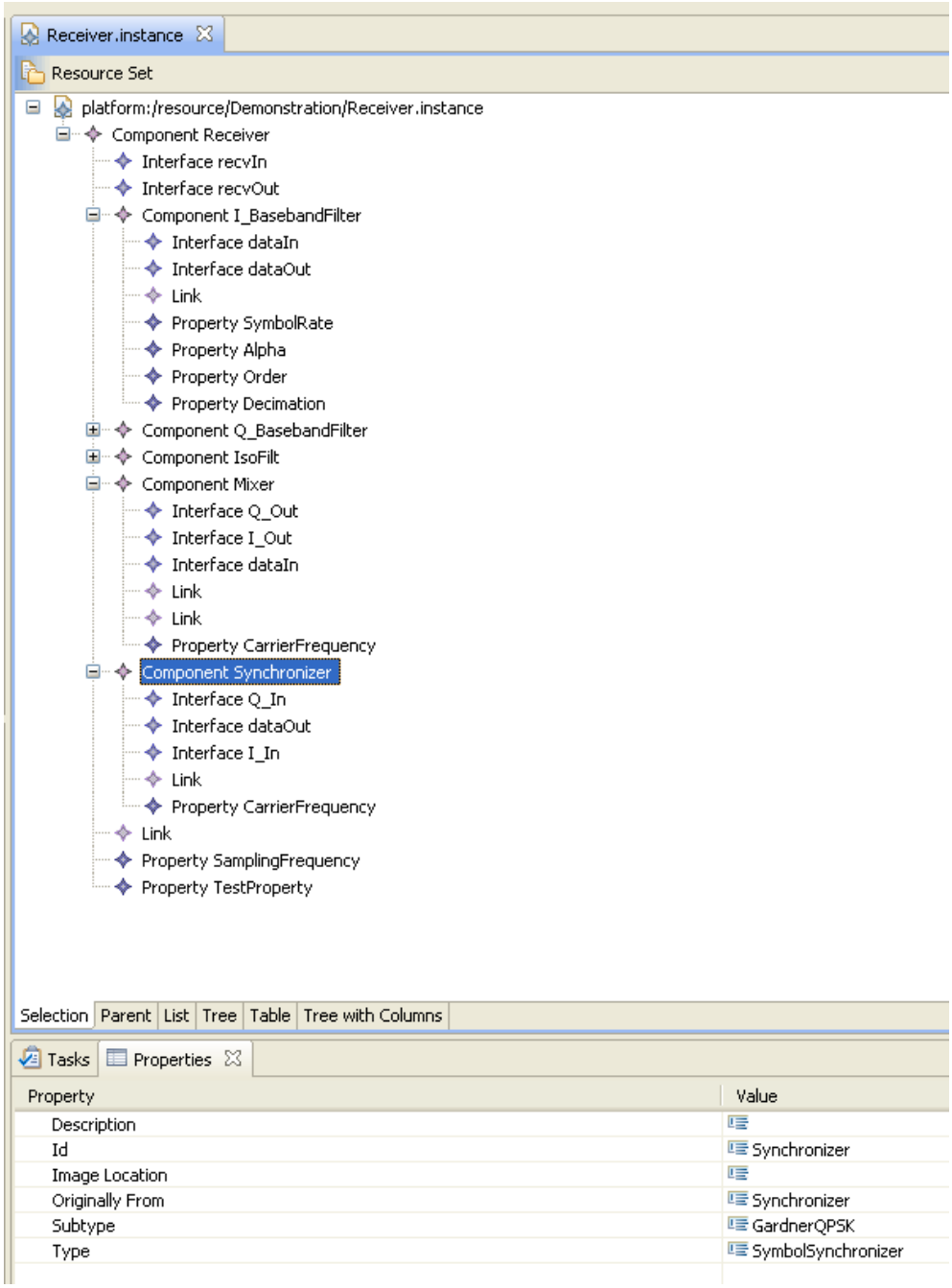

Figure 3-2: Model Editor – Tree View. Users can edit the model directly in the tree view. This lists all of the entities in the model and their properties are editable in the Properties tab.

3.2 Metamodel

The metamodel that serves as a basis for the modeling environment is based on the xADL instance schema components, interfaces, connectors, and links. In the basic xADL instance schema, components, connectors, interfaces, and links are the different elements used to describe architectures [50]. Components and connectors both have interfaces, which are the only ways that data can pass in or out of them. Links connect two interfaces together to show that data flows between them. In xADL, interfaces determine whether they are in, out, or in-out. Links do not have a direction and any semantics that apply to a connection are described by a connector, not by a link.

The basic schema entities in xADL were extended for MDRC. Table 3-1 describes each entity in the metamodel, the fields that the entity includes, and the purpose of the entity. The Eclipse Modeling Framework (EMF) was used to create the metamodel for use in building the graphical editor. The component is the root of the model hierarchy, containing all other model elements along with recursively defined subcomponents. The lowest level components in the hierarchy must have a defined type and subtype, which are used during generation to match up the model with the hardware definition for it. Components also have an image field that lets the user provide a custom scalable vector graphic (SVG) image to represent the component in the modeling environment.

Table 3-1: Metamodel Entities

Entity	Fields	Purpose
Component	Id, Type, Subtype, Image, Interfaces, Subcomponents, Connectors, Links, Properties, Compartments	The root reusable, configurable block of functionality with user-definable properties
Interface	Id, Direction	To define input and output to and from a component or connector
Connector	Id, interfaces	To allow additional meaning to be placed on a connection
Link	Id, Source Interface, Target Interface	To connect two interfaces
Property	Name, Value	To provide user-definable properties
Compartment *	Id, Interfaces, Subcomponents, Condition	To provide a representation for a run-time reconfigurable section based on some condition

* Semantics for partial reconfiguration have not been fully decided, but the environment provides support

The generic component metamodel is illustrated in Figure 3-3, showing how a component is composed of the other model entities. The definition of a component is recursive, so multiple components can be defined or assembled within a component. A component contains connectors, properties, interfaces, and links. The connectors have interfaces of their own. Links do not contain interfaces, but have references to the source and target interfaces.

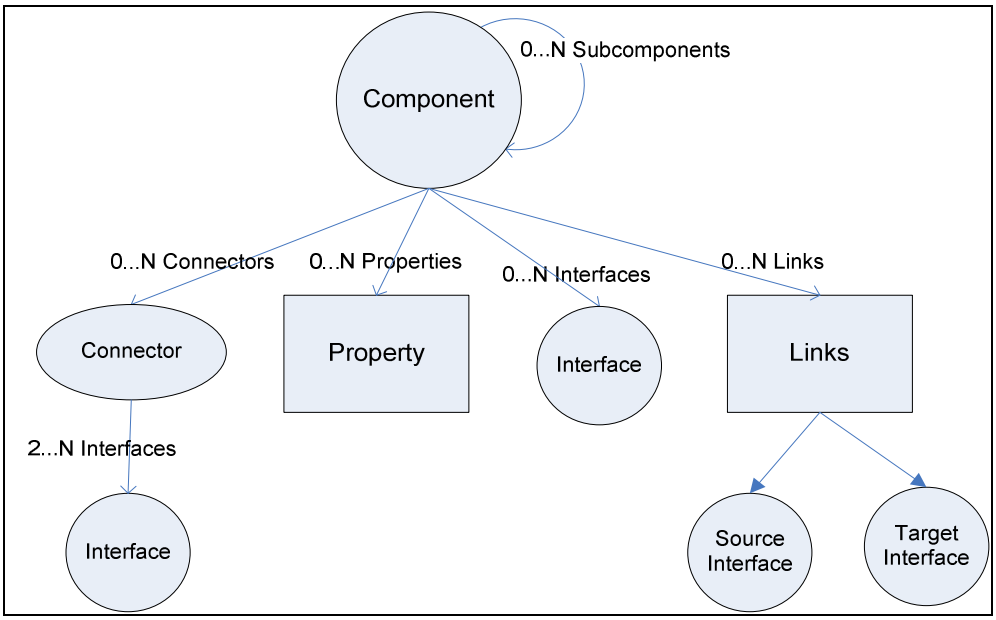

Figure 3-3: Component Model Diagram. The various entities in the MDRC component model and how they relate to one another.

The metamodel entities have the following semantic meaning:

- Component: the component represents a module with user-defined properties. The component has interfaces which are used for input/output and can have all other entities inside of it. The component's properties, type, and subtype are used in the transformation to the PSM.
- Interface: the interface provides a gateway to send or receive information for a component or connector. Interfaces have direction, according to xADL, and define the ports used later in generation.
- Connector: the connector contains any semantics for a connection between components or between another component and a connector. Each connector has at least 2 interfaces, as some kind of input and some kind of output are necessary.
- Property: properties are used to specify any parameters that a component requires. By changing the value of properties, the user modifies how the output module is created during generation.
- Link: the link connects two interfaces, usually the interfaces between two components or between a component and a connector. The link itself does not have any additional semantics, as those are included in connectors, and only maps to a simple wire connection in hardware.

The recursive component metamodel provides for an easy way to use both predefined and custom defined components together. Users are able to build new components out of predefined components and then use those components in other models. When a user pulls out a component into their model, it has all of the default values that were set when the component was created. The user can change its parameters and even drill down into its hierarchy and make any necessary changes. These changes are specific to this instance of the component, and do not have any effect on the original component definition.

Figure 3-4 illustrates the recursive metamodel from the PIM, described by components, down to the hardware module in the PSM. A component is described in the PIM and is generated through the ASM based on instructions that are specific to the target platform. The ASM contains sets of instructions for each component that tell it how to build that component for specific architectures. The instructions may be straightforward HDL, input scripts for a generation tool, or a series of steps using other tools or instructions to build a complex component. The flexibility of the generation facility lets users decide the generation process for their components.

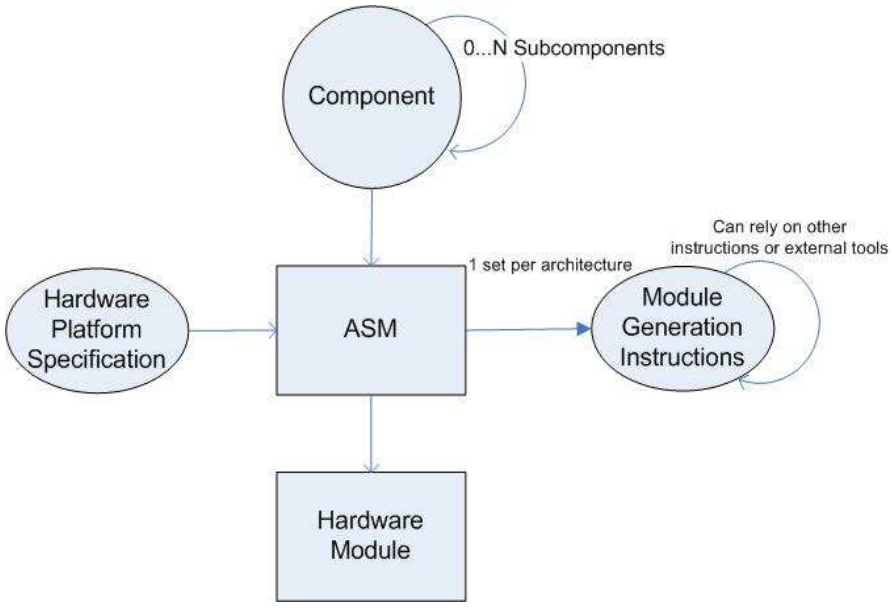

Figure 3-4: Metamodel Diagram. The various entities in the MDRC metamodel and how they relate to one another.

Each predefined component has a type and subtype that link it to a module definition in the ASM. The module specification contains instructions for the ASM to

use during the transformation process to generate the PSM. If a necessary component does not exist and cannot be made out of other predefined components, then the user will need to define it along with the necessary parts of the architecture specification.

3.3 Graphical Model Editor

The graphical model editor, shown in Figure 3-5, is the main interface for creating, editing, updating, and generating a model. The Eclipse GMF project provides the basic tools to create a simple graphical editor around an EMF model within the Eclipse IDE. Additional functionality was added to the basic editor to improve support for recursive editing, custom images, and component reuse. Modifications were also made to add support for generation and property equation evaluation.

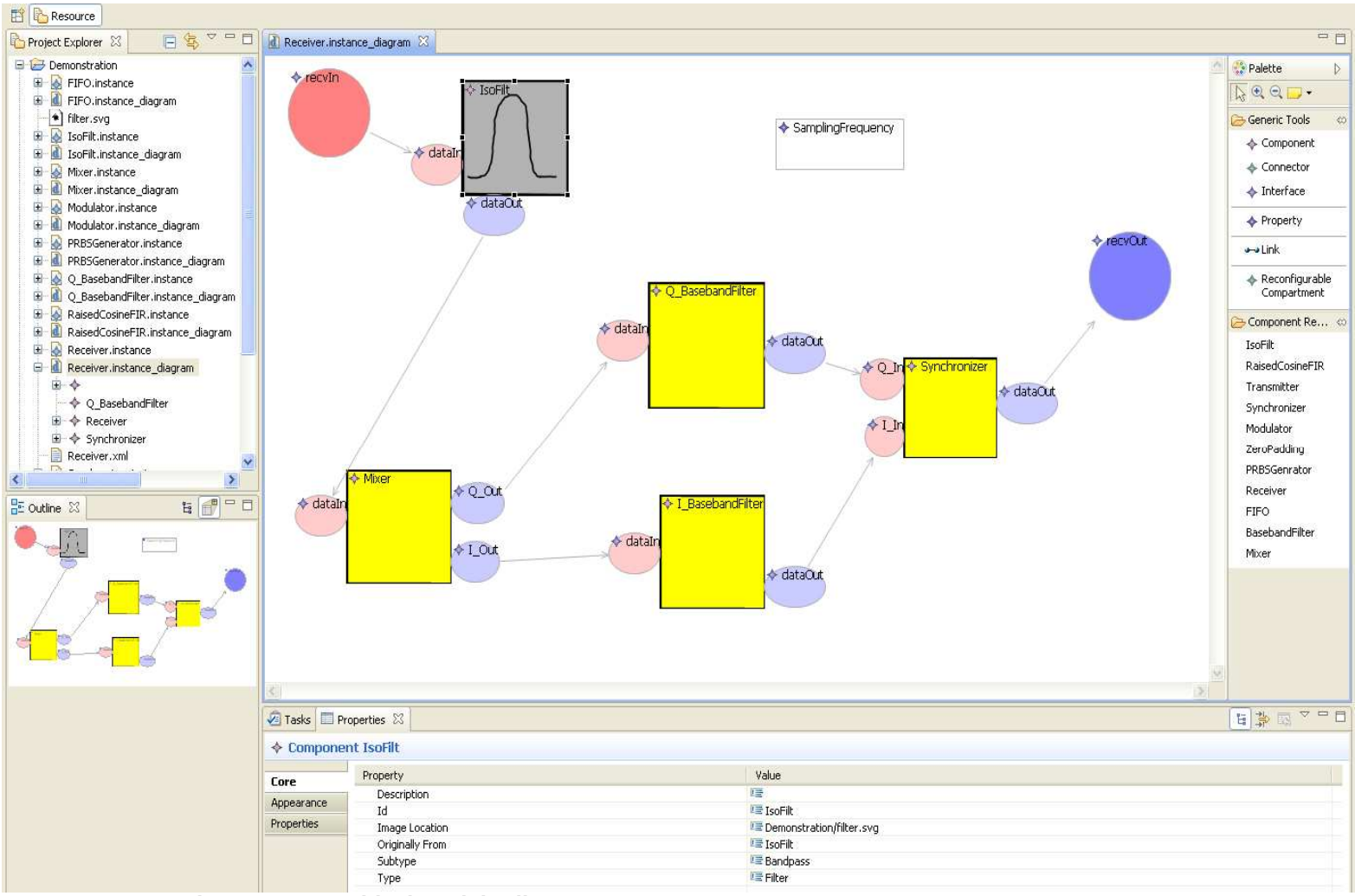

Figure 3-5: Graphical Model Editor. Users can pull out predefined components from a palette, specify interfaces and properties, and connect them together to design an application. The model is independent of the hardware

The user interface was designed to provide a high level view of the application suited for component based modeling. In order to encourage reuse, one of the key goals of the environment, previously made components are readily available from the repository in the design palette. Users can create more complicated components from the predefined ones and then share them in the repository for other developers to use. RC specialists can add new components, for which they have designed the hardware specifications, to the repository to enable others to use them for assembly. The graphical editor is composed of the following elements:

- Model Editor: the main graphical workspace, representing a component. Each metamodel entity the component contains is shown in the diagram space and can be clicked on to view and edit its properties in the Properties view. By double-

clicking a component, the user opens a new diagram window representing that component and everything inside it.

- Palette: contains all of the tools to work with the model editor. The palette can be seen on the right hand side of Figure 3-5. There are two drawers in the figure, Generic Tools and Component Repository. The Generic Tools drawer contains the tools for creating new components, interfaces, connectors, properties, links, and compartments. The Component Repository lists all of the components that are already defined that the user can drag into the open diagram.
- Properties View: a standard Eclipse properties window where the user can view and edit properties of the selected metamodel component. A close-up view of the Properties window is shown in Figure 3-6. The view is tabbed, with the Core tab showing the predefined properties of the metamodel entity and the Properties tab showing user-defined properties. This view is the main method of editing an entity's properties. User-defined properties can be set to equations with other properties higher in the component hierarchy as variables. These equations are evaluated at generation time.
- Project Explorer: the standard Eclipse explorer window where the user can view all of the contents of the project. By integrating the Subversive plugin, the user can link a project to a Subversion repository to download, upload, or edit remote models. Additionally, each model file represents a component which becomes available in the palette for reuse.

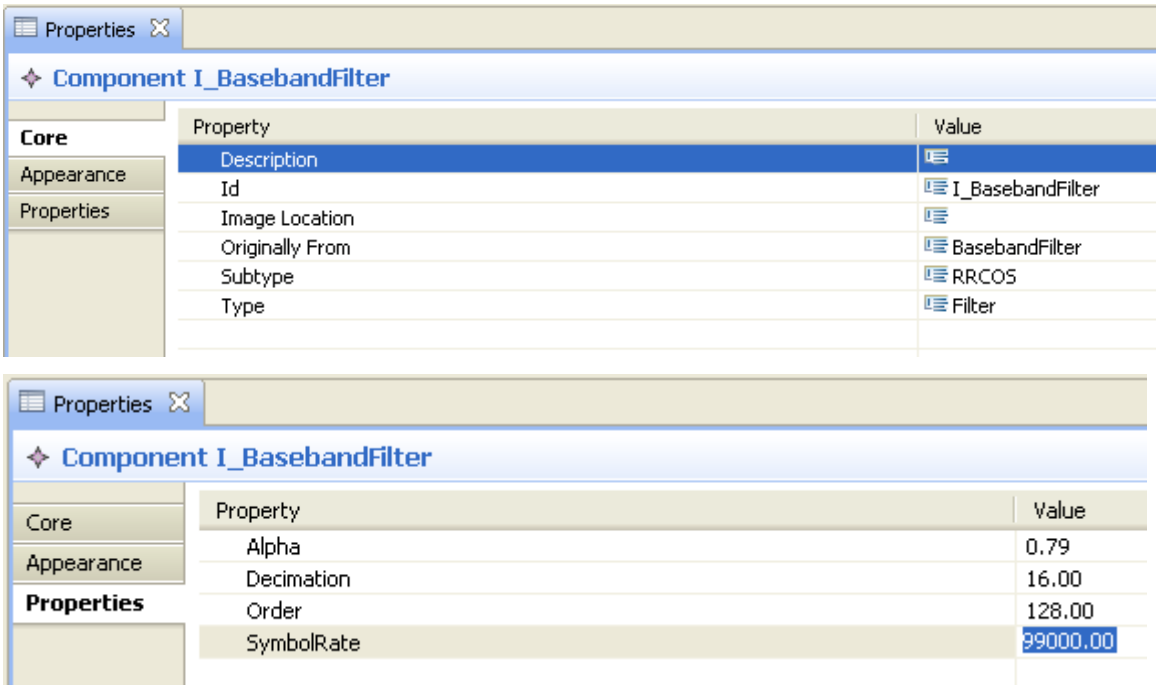

Figure 3-6: Properties View. The window displays the currently selected metamodel entity, allowing the user to view and edit the entity's properties. The view is tabbed, with the Core tab showing the predefined properties and the Properties tab showing user-defined properties.

A few small modifications to the generic graphical editor enhance the model's presentation. Components can be differentiated by custom SVG images, which can be added by the user to the environment. Interfaces are color coded based on their directionality: red interfaces represent inputs, blue interfaces represent outputs, and orange represents in-out ports. Interfaces that belong to the component represented by the diagram are a darker shade than interfaces connected to components within the diagram.

3.4 Generation Framework – The Architecture Specific Model

Model transformation is a continuous process from the high level CIM all the way down to the PSM and code. In software, the process continues on from code, down to a model (machine code) that the machine can understand directly. Just like software, models for hardware applications do not necessarily fit in a limited number of discrete layers. While exploring the application of MBE to the design of a RC system, the researchers found that going from the PIM to PSM for hardware is not a simple, one-step

process. To account for the additional complexity, the idea of an architecture specific model (ASM) was formed.

The ASM, developed by CHREC team members Jorge Suris and Adolfo Recio, is an intermediate transition model used to get from the PIM to PSM layers. Inherent differences in chip architectures make it difficult to reuse components between architectures. Because of these differences, the ASM was created as a standardized way of characterizing how a component is generated for a specific architecture. Figure 3-8 illustrates an example of the information divided between the PIM and ASM for a multiplier component. In the PIM layer, the multiplier's properties are set and its interfaces are connected to other components. The first step of generation is to check if the hardware module is already available. If it is not available, then the ASM describes the steps necessary to prepare the module. In the case of generating for a Xilinx Virtex 4, the process described in Figure 3-7 shows that the Xilinx Core Generator tool [23] will be used to generate the hardware module. The generated module is then connected to other modules with any necessary glue logic.

To handle the process of generating a module, makefiles were chosen. Makefiles provide a well-understood method of running through a series of steps to build a module and its dependencies. In this case, the hardware definition for a component specifies a makefile that holds the ASM instructions for building that component's module. How that makefile decides to build the module is strongly affected by what architecture and platform the system is being built on. All of the details regarding the platform are currently configured in a platform configuration XML file.

The platform specification details the device and chip being used. The FPGA ports are specified and named, which connect the FPGA to the rest of the board. The system clock speed and overall program name are also denoted in this platform XML file. It is important to realize the difference between the FPGA and the board it is on. A board usually has many other components on it, like a GPP or networking components, for carrying out other tasks not related to the logic on the FPGA. The ASM deals solely with the FPGA, abstracting away the details of the platform the chip resides on. If some application logic were to be moved outside of the FPGA, it would need to be detailed in the PSM.

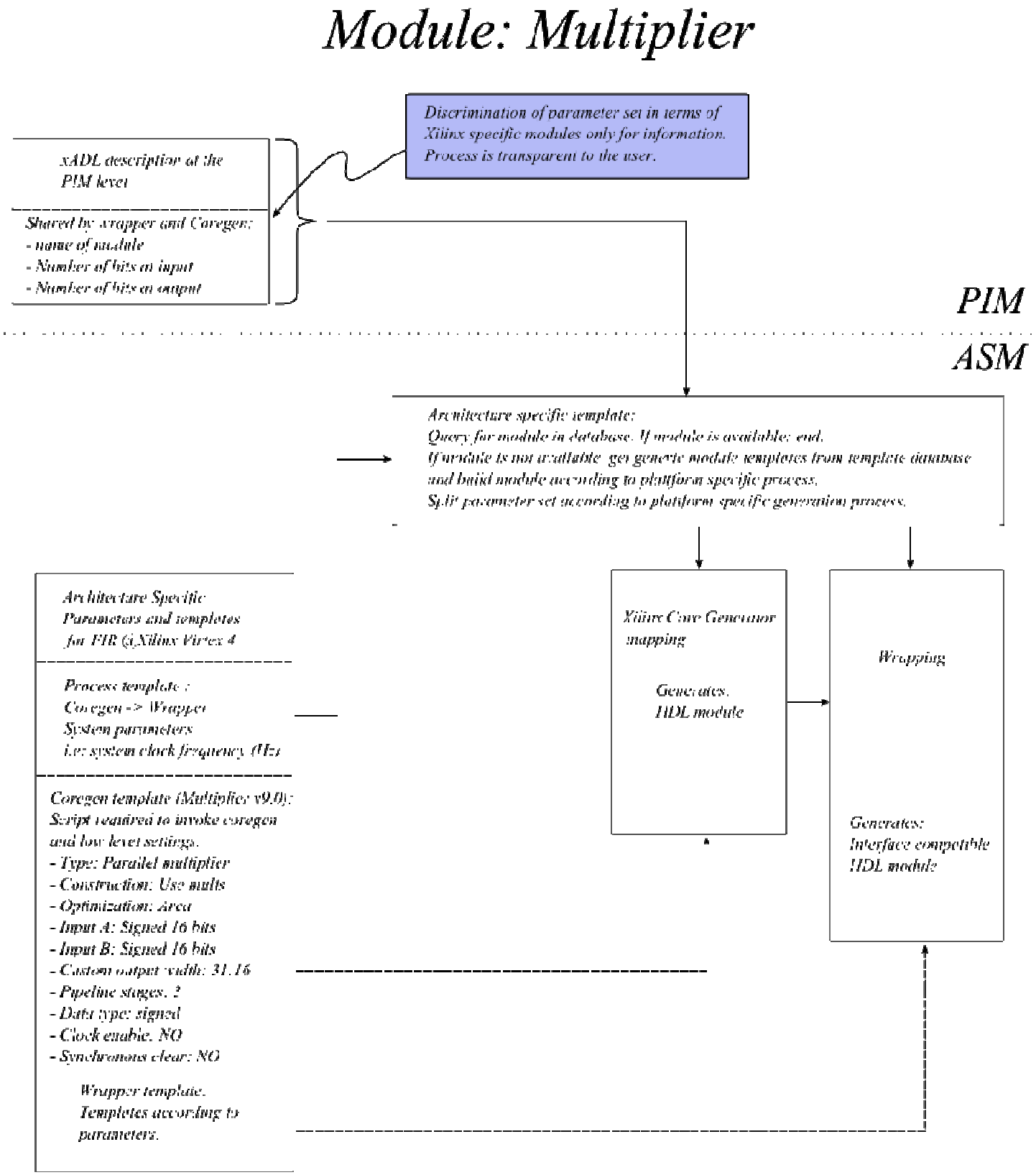

Figure 3-7: ASM for a multiplier. The Architecture Specific Model for a multiplier component, created by CHREC team member Adolfo Recio. specifies how the hardware module is retrieved. If the module is not available, the necessary steps are taken to generate it [51].

The hardware specification XML for a module provides important details to the ASM for generation. This file contains the Type and Subtype that are referred to by the PIM along with the editable properties for the component. The XML file also details any restrictions on the chip or device it can be used with, clock speed, interfaces and ports, and a makefile. The makefile carries out the steps necessary to retrieve or build the module according to the target platform.

The abstractions provided by the ASM hide the complexity of the hardware architecture and platform from the end user. A RC specialist has the ability to add new modules to the ASM and make them available as components in the modeling environment. Instructions for generating a component for different architectures can also be added to the ASM, allowing a module to be used across different hardware platforms.

3.5 Scenario: Building a Transmitter

Software-defined radio is one application area that can receive benefits from a reconfigurable implementation. Throughout the world there are a large number of different standards and protocols for radio devices that present a challenge to the industry [52]. A software-defined radio can be changed on the fly to work with different protocols and carry out different tasks. Advances in reconfigurable technology allow software-defined radios to be implemented in hardware to achieve real-time performance. This is made possible due to the flexibility provided by a reconfigurable device.

In order to demonstrate the use of the MDRC modeling environment, the CHREC team set out to build a transmitter to go on a FPGA. The design process involved first coming up with a platform independent model to be represented in the modeling environment. To accomplish this task, the necessary components were built and added to the model repository. Figure 3-8 illustrates the high level design of the transmitter. The goal was to show how a transmitter could be built from premade components and its parameters could be easily changed.

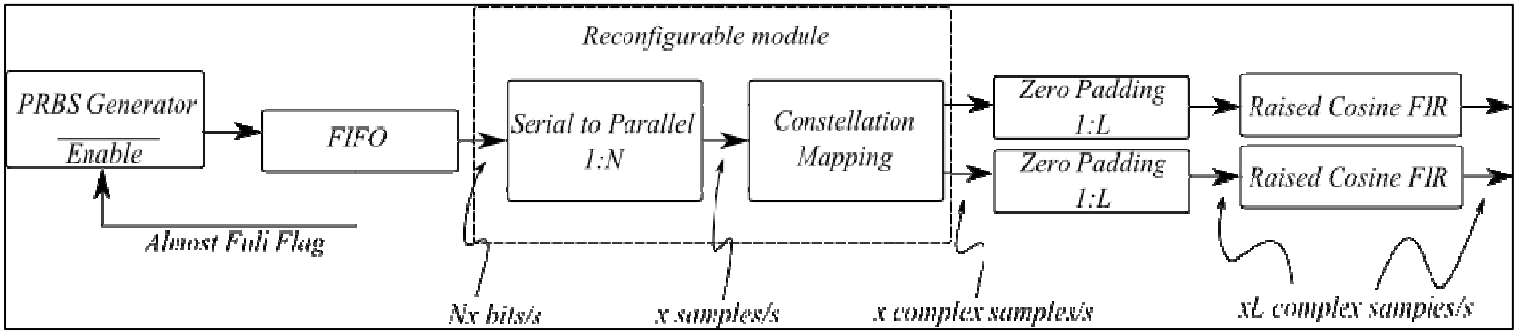

Figure 3-8: Digital Modulator High Level Design. The high level diagram for a transmitter, created by CHREC team member Adolfo Recio [51]. The modulator can be reconfigured to a different modulation type based on parameters.

A transmitter is a device that sends out a radio signal by using modulation. Modulation is the process of preparing a signal to carry information and be sent over the radio. The CHREC team's transmitter design contains the following components:

- Psuedorandom Bit Sequence (PBRS) Generator: generates the bit stream that will go through modulation. This component is advantageous for testing because a similar setup can be made at the receiver for error-checking on the communication system.
- FIFO: the first-in first-out queue controls the flow of bits from the PBRS generator
- Serial to Parallel: part of the modulator. This component converts the groups of bits so they can be mapped into the points of the constellation mapping
- Constellation Mapping: part of the modulator. This component contains a lookup table where locations on the complex plane are assigned to the bit words that come from the serial to parallel converter. Changing the lookup table changes the modulation type.
- Zero Padding/Upconverter: pads the complex sample result of the modulator so the raised cosine FIR can shape the spectral response of the modulated signal.
- Raised cosine FIR: a filter used to shape the modulated signal after it has been padded.

Figure 3-9 shows the transmitter design represented in the modeling environment. The diagram in Figure 3-10 shows the subcomponents contained in the Modulator component.

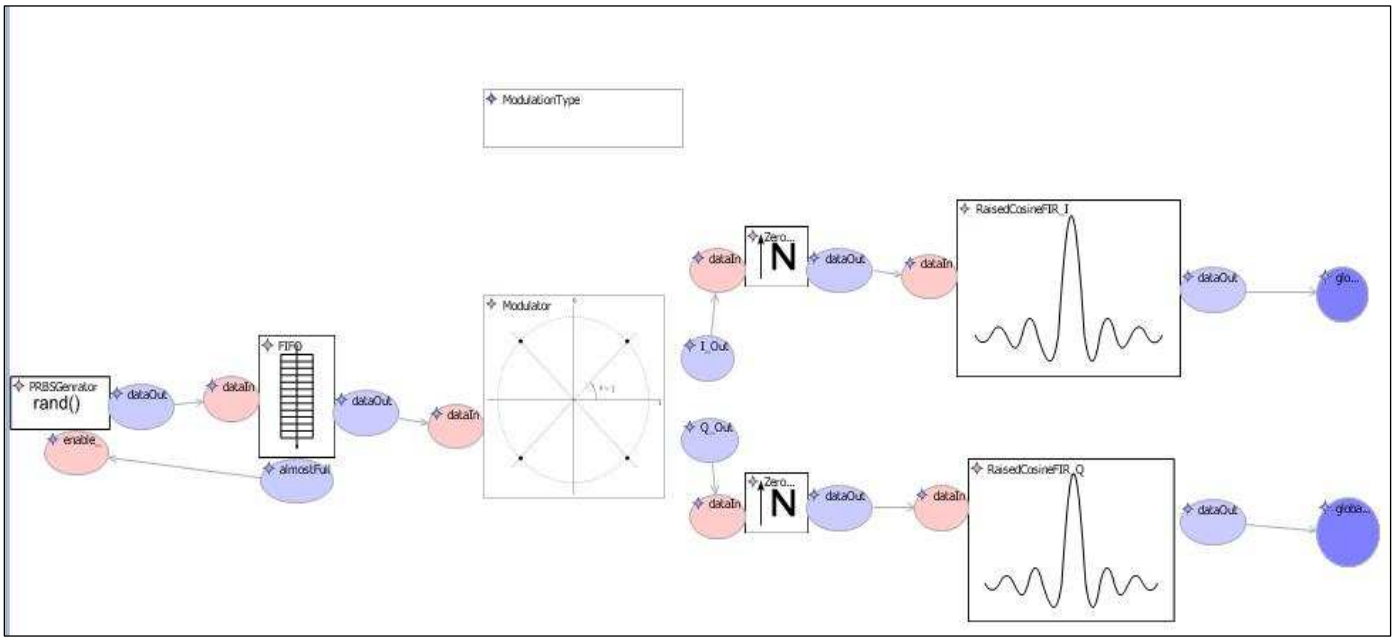

Figure 3-9: PIM for a Transmitter. The platform independent model for a transmitter represented in the modeling environment.

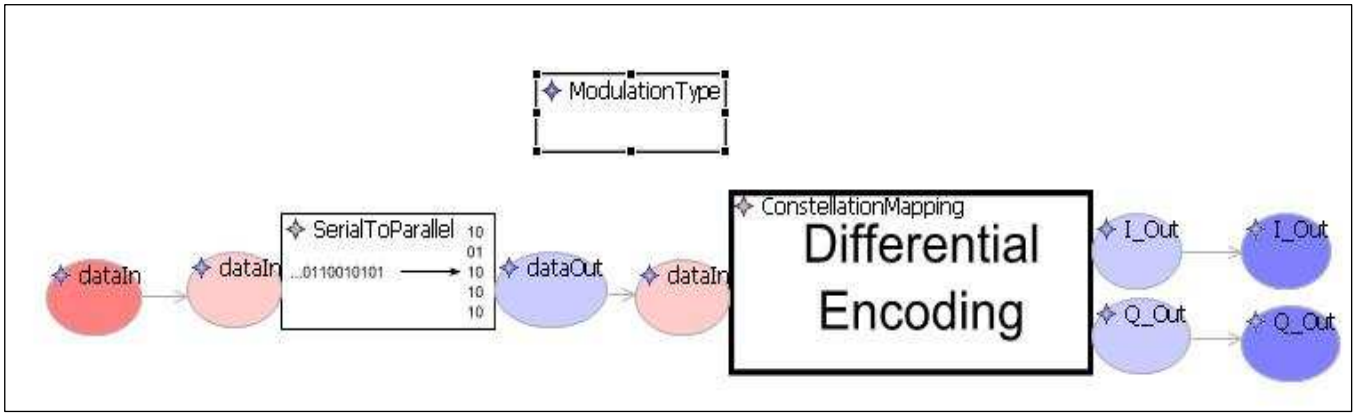

Figure 3-10: Modulator Component. The contents of the modulator. The modulator contains the serial to parallel converter and constellation mapping.

The property in the transmitter called ModulationType determines the number of bits per symbol that will be used for the modulator. The modulator component and its subcomponents all draw their modulation type property from the top level transmitter's property. The value of the modulation type will be used during generation to determine how modulation is implemented.

The first step to building the transmitter in the modeling environment was building all of the components that would need to be used in the transmitter design. For each component, a new diagram is made and any interfaces, properties, and

subcomponents (in the case of the modulator) are put inside, named, and given default values for their parameters. The components that are backed by hardware modules have parameters that let the ASM search for that component for a specific architecture. After the components are saved, they are made available in the palette for reuse in the transmitter component diagram. Dragging out each component, setting its properties, and linking up its interfaces were relatively simple tasks.

Once the transmitter has been fully specified in the modeling environment, it can be generated. The first step in the generation process produces a XML representation of the PIM. This contains the transmitter's modules, their parameters, and the connections between them. In the case of the modulator component, only its subcomponents actually represent modules. Only the leaf components in the hierarchy are generated and the connections are carried through to the appropriate component interfaces. In this example, the dataOut interface of the queue maps directly to the dataIn interface on the serial to parallel component, since that is the information that needs to be passed on to the platform generation facility.

In order to fully generate the implementation, the user must provide a platform specification file. This XML file gives the generation tool the necessary information regarding the architecture and platform the PIM is being generated on, along with the ports that will be used on the board. With this file included, the PIM can be sent to the ASM for generation. As described in the previous section, the generation facility builds each module according to the instructions in the ASM for that module for the specified architecture. In the demonstration, the transmitter was built for a Xilinx Virtex4, whose details were specified in the hardware platform XML file and the components' build instructions were specified in the ASM.

The ease of modifying the design is demonstrated by changing the modulation type. By changing the top level property in the transmitter, changes are propagated down to the modulator and its subcomponents during generation. When it reaches the ASM, the correct rules are followed to generate the modules based on the new value of the parameter.

Chapter 4
Analysis and Discussions

The previous chapter detailed the MDRC approach to reconfigurable system design. The modeling environment provides a way to work with hardware applications at a platform independent level, separate from the implementation. The MDRC prototype showed the process for modeling a transmitter and generating it for a Xilinx FPGA.

This chapter examines what was learned while conducting this research with respect to the modeling process and hardware design. The model-based approach offers many advantages to the hardware application design process that can be achieved through higher levels of abstraction. The feasibility of using MBE with RC ultimately comes down to the tradeoffs that occur from raising abstractions and being able to deal with the unique problems that come with abstracting hardware.

In order to fully integrate MBE into the reconfigurable design process, several challenges need to be overcome. Along with these challenges are many opportunities to harness the potential power of reconfigurable devices and their ability to be reconfigured. The MDRC environment is an early prototype designed to show the basic capabilities of MBE in the RC environment. This work was completed over the past year working with the NSF Center for High-Performance Reconfigurable Computing (CHREC) team. The specific focus of the research completed in this thesis is on the feasibility of and experience with the MDRC environment. The demonstration prototype provided several opportunities to experiment with the MBE concepts as they were employed on Software Defined Radios (SDR). The Electrical and Computer Engineering (ECE) members of the team focused on the SDR domain models while the Computer Science member of the team focused on the environment to demonstrate the MBE in terms of modeling, transformations, and code generation.

While the prototype does demonstrate the feasibility of MBE at the essential levels, many places exist for changes, updates, and additions to the modeling environment to continue to improve its ability to help developers design reconfigurable devices efficiently.

As you may recall from earlier, the hypothesis that this research sets out to examine is if MBE can be effectively applied to the field of RC to reap some of the benefits that software engineering has drawn from a model-based approach. The research aims to raise the abstraction level for RC development, leading to an increase in productivity. As seen in software development, increased productivity was made possible by high-level abstractions that resulted in a separation of concerns between application logic and implementation and increased developer ability to reuse work products.

The key research questions that stem from this hypothesis are:

- Is using MBE for FPGA development feasible?
- Can reuse and portability be supported?
- Are productivity gains possible by using a model-based development environment?
- How do these compare with similar gains/losses in software engineering?

To effectively analyze and discuss the results of this research the following points of evaluation will be highlighted which support answering the research questions: construction of a modeling environment, reuse of components, portability between architectures, and productivity tradeoffs. The MDRC framework provides a modeling environment for users to develop RC applications with high-level abstractions. Reuse, which strongly affects productivity, is enabled through an extensible repository. Platform independent modeling and automatic generation support portability between platforms. Performance at run-time is traded for performance at design time. Each of these points is discussed in more detail in the following sections.

The basic research approach is to develop a modeling environment where MBE principles can be applied to RC design. Users will create a high-level PIM in the environment based on the requirements of their system, which are detailed in their own CIM. The PIM is used to generate the application for a specific architecture based on details in the ASM for the components they used. The ASM helps bridge the gap between the PIM and the desired implementation, or PSM. The PSM will vary depending on what platform the application is being generated for. In the case of a

FPGA, it would be the implementation HDL tooled for the architecture of the chosen FPGA.

The next section will analyze and discuss the results of the transmitter scenario described in Section 3.5. The advantages of the model-based approach and its relation to RC and the transmitter scenario will be discussed in Section 4.2. Section 4.3 explores some of the unique challenges that face applying MBE to hardware. The feasibility of meeting these challenges and being able to use MBE for RC on the long-term is discussed in Section 4.4. Opportunities and recommendations for future work on the MDRC framework, and MBE for RC in general, are explored in Section 4.5.

4.1 Scenario Results

To demonstrate the abilities of the MDRC environment, a transmitter was modeled and generated for a specific modulation type (Q-PSK). The example application was for a digital receiver with a 100 KHz symbol rate, a 1 MHz carrier frequency, and differential encoding.

The design process starts with the requirement for a transmitter. The transmitter is specified at an abstract level in the software-defined radio domain and a number of possible designs exist in the SDR application space to meet the requirement. From those designs, the RC engineers selected one for a digital transmitter with Q-PSK modulation. The selected transmitter was illustrated at the PIM level in the modeling environment, based on the requirements set forth in the CIM. Each component was modeled with the correct properties and interfaces, and then connected appropriately to achieve the desired functionality.

The original modulation type was Q-PSK and part of the scenario was to show that the design could be changed to support 8-PSK and the new transmitter rendered automatically. By changing the modulation type property in the editor, the resultant generated system design was also changed. This is made possible through the generation of the model from the modeling environment and the transformations provided by the ASM. By taking advantage of the abstractions and generation capabilities, the user was able to reuse the design to create a transmitter with a different modulation type with little

effort. In this case, the user is only changing the properties of the PIM components, reusing 100% of the high-level transmitter design. Even though the property change may result in a change in hardware, requiring a new version of a module to be generated, the end user does not need to worry about it. The ASM contains all of the necessary information to generate the hardware module for any of the architectures that component is available on.

In the case of the modulator, the component is actually composed of two other components, a serial to parallel converter and a constellation mapping. Each of these components has a property for modulation type, which represents the bits per symbol in the modulation. For the serial to parallel component, this affects what happens to the bits before they are sent to the constellation mapping. For the constellation mapping, this affects its input as well as the lookup table that is used to determine its output. Based on the rules in the ASM, the correct modules are generated for implementation.

Additional properties were made available for the user to change in the modeling environment. Table 4-1 lists the major parameters that could be modified before generation. By having editable properties that have an effect on the generated modules, many different versions of the components can be built without needing to redesign or change any code.

Table 4-1: Configurable Transmitter Component Parameters

Component	Parameter	Description
PBRS Generator	Polynomial	An equation that determines the bit sequence generated.
FIFO	Size	The amount of data that can be stored in the queue
Modulator	Modulation Type	The number of bits/symbol used in modulation
Raised Cosine FIR	Upsampling Parameter	Defines the quality of the spectral shaping that can be obtained at the modulator.
Raised Cosine FIR	Roll-off factor	Defines the bandwidth used by the modulated signal.

Taking advantage of external tools to help generate modules significantly reduced the amount of work necessary to make a component available. The Eclipse environment

and related frameworks were heavily leveraged to provide users direct manipulation of their models and quick and easy reuse through a repository. Since the models are backed by XML, the contained information is easily sent to other parts of the MDRC framework during generation without the need for any user interaction. The makefiles in the ASM take the model from the modeling environment and use it to generate the implementation.

The makefile structure allows calls to many different external tools, as long as those tools have an accessible API. Most of the components in the transmitter scenario were generated with the Xilinx Core Generator. The flexible nature of XML allows for quick and easy transformation between formats to prepare data to be sent to the tools. Figure 4-1 describes the ASM for a low-pass FIR filter. The ASM for the filter is fairly complicated, requiring multiple steps to generate the module for the Xilinx Virtex 4. The Xilinx Core Generator requires the filter coefficients along with other parameters to generate the filter. In order to get the coefficients, MATLAB is used to actually design the filter. To make calls to these two external tools, the ASM first takes the XML data from the PIM and calls a MATLAB script using some of those parameters. The output of the MATLAB script is put in a coefficients file to send to the Xilinx Core Generator, which outputs the hardware module. Any wrapping or necessary glue code to hook up the interfaces is then completed. All of this is taken care of by the ASM, though, so the front-end user is only in charge of filling in the necessary parameters at the PIM.

Module: Low Pass FIR Filter

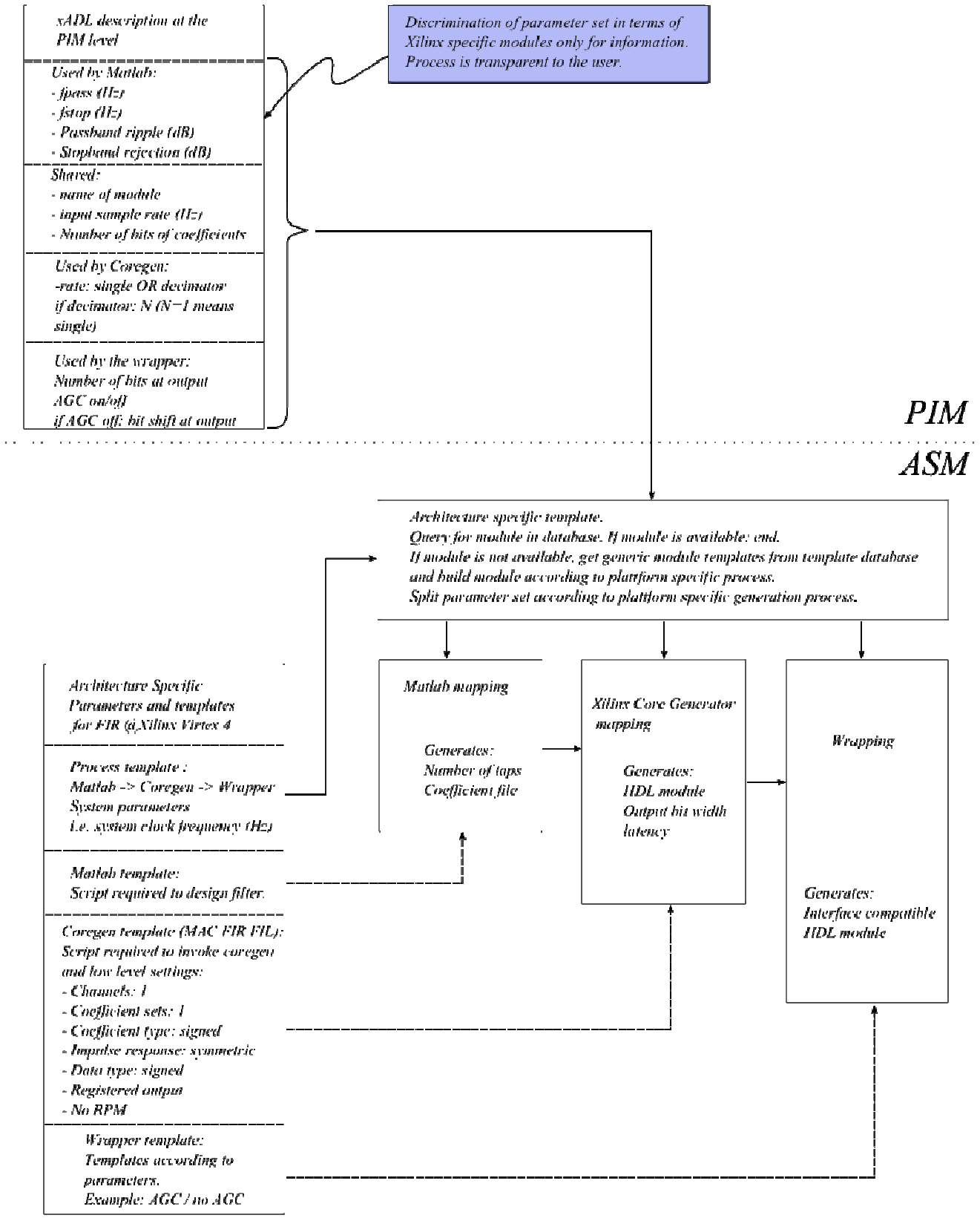

Figure 4-1: ASM for a FIR filter. The architecture specific model for a low-pass FIR filter created by CHREC team member Adolfo Recio. The ASM contains all the steps necessary to generate the module for a specific architecture [51].

Utilizing external tools to assist in building modules relieved some of the burden from the RC designers on the team. At the same time, using other tools to help in the generation process works towards one of the major goals of the research, reuse. Leveraging other tools made the process of adding new components to the repository easier, increasing the overall productivity of the system. The effort required to implement a new component is significantly higher than the effort needed to use that component in a design.

The research team found that modeling the high level design for the transmitter took a small amount of time compared to the time that went into making those components available. Several hours were required to initially create each component by deciding on functionality, how the component would be generated, and what parameters would be changeable. Less than an hour was needed to put the design together in the modeling environment by selecting and connecting the right components. Once a design has been picked, modeling the design at the platform independent layer is a simple task of selecting the right components from the repository, connecting them correctly, and configuring their parameters as desired.

From the premade components that are available, more complex components can be assembled. Now that a transmitter component has been created, a user can reuse the transmitter in other designs simply by dragging it out from the repository, setting its properties, and connecting its interfaces. Additionally, if a new transmitter needs to be designed, many of the components will already be available for it. This cuts the design time for subsequent projects significantly because of the components that can be reused. Instead of several hours for multiple components, only one or two new components may be necessary in the new design.

Given that RC entails reconfiguration, the need for some way to provide run-time reconfigurablility for a system was necessary. One of the main advantages of FPGAs is that they can be reconfigured to do another task, unlike ASICs. Current research in reconfiguration is examining efficient ways to do partial reconfiguration [42]. Partial reconfiguration is when only part of the FPGA circuitry is changed while the rest stays intact. Since a FPGA can be reconfigured at any time, part of the device can be changed to carry out another task while that part is not being used. In the area of software-defined

radio, an example of this might be switching to a different protocol while the radio is running to transmit or receive a different kind of signal. In the case of our modulator, it would be convenient to be able to specify two different configurations for it and a condition that lets the device know when to switch between them.

In the modeling environment, this can be represented by using a reconfigurable compartment. The compartment somewhat resembles a component in that it has interfaces and contains subcomponents; however, a compartment's subentities are visible in the current diagram. Figure 4-2 shows the model diagram for a reconfigurable compartment. The component shown has only a property, two interfaces, and a reconfigurable compartment. Inside the compartment, there are two different receivers with different sampling frequencies, each connected to the compartment's interfaces. When this diagram is generated, the XML for the model will describe the different configurations within the compartment and the condition provides the rule to decide which one is used when.

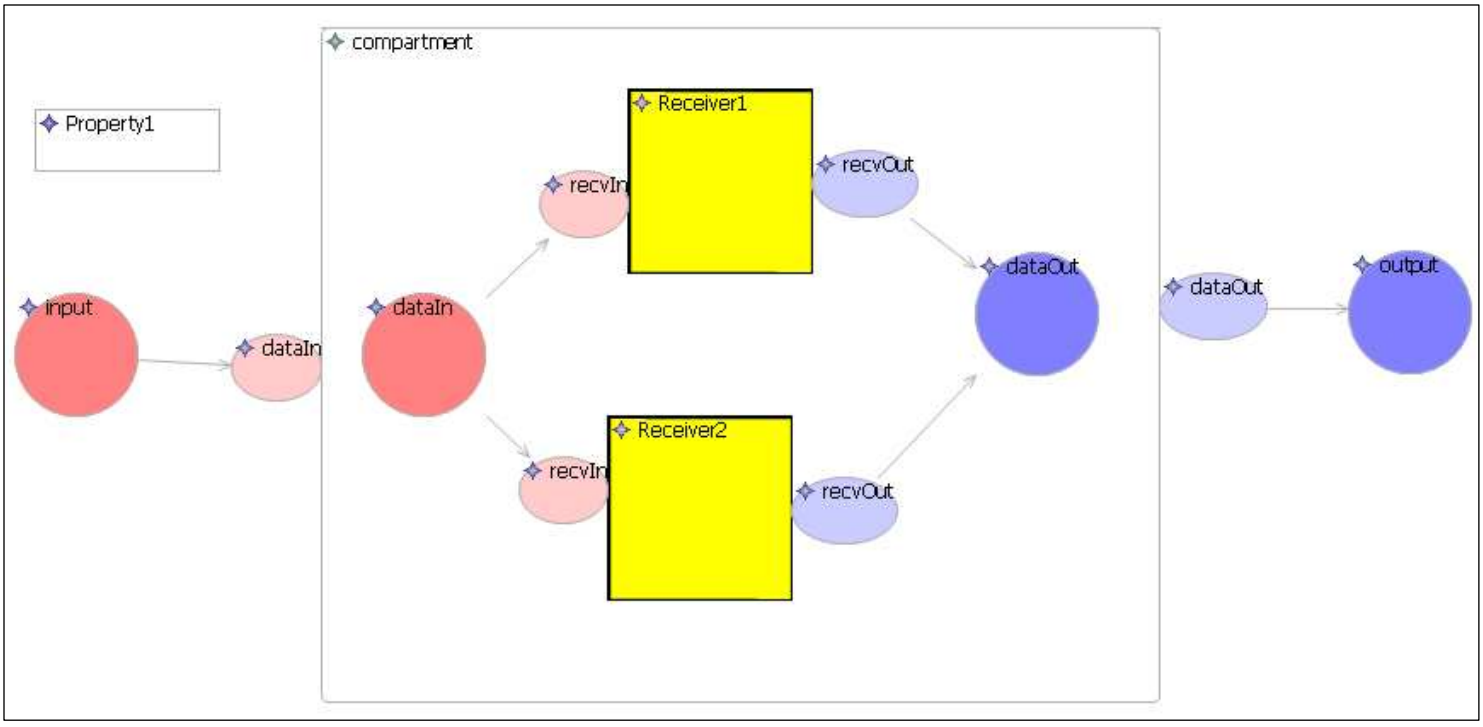

Figure 4-2: Reconfigurable Compartment Example. The model design for a reconfigurable compartment. In this example, there are two receivers that are available. The decision on which one to use is based on the compartment's condition property.

While the first generation of the MDRC facility does not provide full-support for partial reconfiguration, the importance of this model is that there is a platform independent way to represent partial reconfiguration. Different architectures may provide completely different implementation strategies for partial reconfiguration or it may be done in a completely different manner in the future, but there is a higher level model that can be separated out from those implementations. That separation of concerns is one of the key values of the model-based approach. If new research in partial reconfiguration is made available, it can be plugged into the MDRC framework.

The research team made several key observations and learned many lessons while working through the demonstration scenario. It became clear during the design process that the majority of time was spent on developing components to put in the repository. Since the project started with a completely blank repository, there was nothing available to reuse. The model-based approach requires developers to put a large amount of time and effort up front in order to be rewarded later on in future releases of an application or for future projects in a domain area. Once components are available, it is much easier to build the system and customize it to the requirements. Creating elegant, reusable components requires more time than just building them for instant use in a project, but can have a greater impact in the future.

The research team learned that MBE may not be appropriate to use for a one-time project in an unrelated domain area. If the requirements are straightforward and it is unlikely that many changes will occur throughout the project's life cycle, the MBE approach may not do much for the productivity of that project. However, if a future project in the same domain area can reuse some of those components, then it would increase the benefit of using MBE in the beginning.

Another observation was how beneficial a flexible ASM was for leveraging external tools and enabling portability. Since the ASM is not tied to any particular format, some components made use of tools like MATLAB and Xilinx Core Generator to create hardware modules. Using these tools allowed the developers to build on the work of others to provide quality modules for implementation. The flexibility left room for other generation methods to be used to build the system on varying platforms.

Taking advantage of the reconfigurable part of RC is one of the long-term goals of the model-based approach. Reconfiguration is what sets RC apart from other hardware, but the research team found it is difficult to represent it in a modeling environment. Generating two different configurations is not a hard task, but accounting for partial run-time reconfiguration is challenging. Integrating other research on describing partial reconfiguration seems to be the best approach for providing that functionality in the MDRC framework.

The research team observed that MBE presents a number of tradeoffs to the developer. More productivity in the design process is achieved by sacrificing some optimizations that can be made to run-time. Initial development costs are increased in order to bring down costs in the future through improved reuse. Better abstractions allow developers to reason about problems at a higher level, but they lose the ability to make some customizations. How well these tradeoffs are implemented will determine if the advantages they provide are enough to make MBE a feasible approach to RC.

4.2 Advantages of the Model-Based Approach

Model-based engineering provides many advantages to the software development process: increased reuse, platform independence, automation, traceability, and change tolerance. This research set out to directly apply MBE principles to reconfigurable development to see if these advantages carried over to hardware. The CHREC team found that the prototype MDRC modeling environment provides the ability to reuse hardware components, the ability to generate for multiple platforms automatically, and limited traceability. Higher level abstractions are a key factor in raising productivity, making the improvements in reuse and portability to the design process possible. The abstractions allowed a user with little RC experience to put together and customize a transmitter knowing only the high level design details and not the inner workings of the components.

Increased reusability of components is readily apparent when using the modeling environment. Already assembled components are available to the user to use in their own design and modify as they see fit. Additionally, any component the user creates can be

added to the library for later reuse. The component repository provides a way to collaborate and share experience.

The transmitter that the CHREC team built for the demonstration scenario provides a reasonable example of reuse. The version of the transmitter that runs off a random generator is currently available in the component library. This version is really only useful for testing purposes, but what if a user wants to connect it to some other source to transmit a signal? The user can drag the transmitter out of the component repository and replace the generator with an input interface. Next, the user can connect some other signal source to that interface in their component diagram. This quickly allows the user to reuse the capabilities of the transmitter but with their own signal source. If only the signal source is changed, then 7 of the 8 leaf components have been reused for 87.5% reuse of the previously made transmitter. It is unlikely to expect such a high degree of reuse of whole designs every time. However, as the component repository grows, it is reasonable to expect that a majority of components will have already been developed and can be reused. If a new developer wants to completely change the modulator and signal source, but use the same queue and filtering, they could reuse 62.5% of the demonstration transmitter. This level of reuse would still have a large impact on productivity as developers would not have to go back and redesign every single component, which is very costly. If the desired components are already in the repository, then the developer can achieve even higher reuse. With this level of reuse, design time has switched from being about building the individual components to building the system by finding the right components and connecting them correctly.

The platform independent model of the application goes a long way towards improving portability. Portability can have an effect on productivity in a couple different ways. First, in the case where an application is being targeted to several different platforms, portability saves on development costs. Consider the scenario where a receiver is being built and several instances of it will be put in three different locations. Two locations have different hardware available and the receiver at the third location will be implemented in software. Without any easy way to port the application to these three very different platforms, three whole development teams will be needed to build the application tripling the development costs. If one version of the application can be

developed and then generated to those different platforms, a single development group can be used because of their increased productivity.

The second major productivity increase due to portability can be seen when sudden changes in system requirements or technology occur. If a group of RC engineers has been writing a hardware application using a C-mapper that targets only one specific platform, they will be in trouble if a new requirement or business decision is made that requires them to switch platforms. Additionally, if better technology is created in the future, but is not compatible with older versions, it will be difficult to port code from the old version to the new one. Accounting for these changes by making the system portable will greatly reduce development costs in this situation.

In the transmitter scenario, the application was only generated for a Xilinx FPGA for demonstration purposes. The generation facility framework offers the opportunity to generate for any other platform as well. If the CHREC team had the design tools to work with other architectures, versions of the components could also be made available for those platforms. If those other versions are available, then a user simply changes the generation parameters in the platform specification file and the system will automatically be generated with the correct modules.

A platform does not have to necessarily be a FPGA, or even hardware. Because each ASM contains its own rules for generating the module, the resultant PSM could be in any format, even code for software. Over time, requirements change, business decisions change, and technology changes. Each of these changes has an impact on application development, sometimes requiring the system to be implemented on a completely different platform. If developers have to start over because of one of these changes, then productivity suffers, especially if all of their design artifacts are closely tied to the implementation platform. Using MBE can solve this portability problem and help the system deal with changes throughout its life cycle.

The only effort needed to generate the PIM in Java instead of HDL is to provide the proper transformation rules that take the model and generate Java code. This level of portability gives application developers significantly increased flexibility. Since the application is not tied to the implementation, it is relatively easy to switch between platforms as necessary, or even to try out the application on different platforms to test the

differences in performance. Considering the parameters that can be set on reusable components and the flexibility of generating the system to different platforms, the number of possible solutions to a problem that can be achieved is far greater than what could be done through manual design.

Due to the flexibility of the generation facility, developers can work with as high or low abstractions as they desire. The transmitter developed in the demonstration can be considered a higher level component, since it is based off of several other components. However, if a developer so desired, they could create components for individual logic gates and have it link directly to HDL. This goes against the ideals of MBE, but shows the flexibility of the system. Additionally, the programmer can dive into the generated code in order to make very specific low level changes. This is not recommended, as generated code is usually unfriendly to humans due to the optimizations and changes that a compiler can apply. So, there is a tradeoff for using MBE between initial development costs and future productivity. Instead of building components from very basic building blocks each time, developers using MBE spend the time early on to make high-level, highly reusable components. Time is saved in the future by taking advantage of reuse and integrating these components into a solution, ultimately leading to increased productivity.

The model-based approach provides the opportunity for a large number of helpful design tools to be integrated into a unified development approach. In the demonstration, components are generated through a mixture of MATLAB scripts, Xilinx generation tools, and custom built modules. The system's flexibility allows for practically anything to be used as the basis for a component. The significant contributions from research on hardware compilers can be leveraged in the model-based approach. A developer might design a highly optimized module in a C-based hardware language and then make it available in the component repository. Various commercial vendor tools can be integrated to provide quick module development within their architectures, as was shown in the demonstration. Hardware patterns and templates can be codified as components that have parameters or subcomponents that need to be filled in before generation. The model-based approach does not attempt to replace any of these research areas, but offers a way to integrate them all into a unified development environment.

Reuse and portability are not the only areas of development that MBE enhances. Due to the inherent nature of having models at different abstraction layers, the model-based approach gives traceability for little additional effort. A full model-based approach begins at the computationally independent level, before the system designer even considers the system. This is the stage where the requirements and constraints of the application are decided. When the PIM is created, the requirements are closely followed to represent a method of computationally achieving the desired functionality. When the PSM is generated, those requirements can be traced down to the implementation that carries out that functionality, ensuring that it brings about the desired result. Moreover, reuse at the higher levels means that a developer does not need to relearn the domain and requirements in future solutions. For example, consider the case where support for a new platform is being planned for a future release of a system. The design work that was already done for the system may still be quite viable and all of the documentation, test cases, and other development artifacts may still be relevant. These work products can be reused in the new system while the implementation on the new platform is generated based on the new design specification. The developer did not have to go through the trouble of relearning the intricate inner workings of the system to make this upgrade, increasing productivity.

Models are also free documentation for developers. In many development projects, documentation is an afterthought and models that were initially created for designing the system concept are never updated [53]. When using MBE, models are the center of the design process and always provide an up-to-date look at the current state of the system. Additionally, the modeling environment provides version control through the Subversion repository. Developers can look back at older versions of the system and see the changes that were made and how the system has evolved over time.

Software development has been about trying to find effective ways to deal with change throughout the history of software engineering. Systems will continue to evolve over their life cycles and their complexity will continue to increase if nothing is done about it [54]. As hardware becomes programmable and changeable through reconfiguration, it will also need to account for evolution [2]. The model-based framework offers a standardized way to deal with change. The separation of concerns

between logic and implementation divide the problem space into localized areas of change. The logic is encapsulated from any changes that occur at the lower implementation levels and the components are always available for reuse regardless of how the higher level architecture changes.

All of these opportunities have major potential for increasing productivity of the reconfigurable design process. A model-based approach to RC can bring about new levels of reuse and integration to hardware development. In order reap the benefits of all of these advantages, several challenges unique to hardware must be overcome.

4.3 Challenges

Considering that reconfigurable hardware and software are based around different architectures, it is not surprising that hardware design has several unique challenges that are not present in software development. While software is set up to work through a set of general instructions over time, hardware solutions can take advantage of spatial layouts to parallelize tasks [54]. Hardware designers worry about how many gates they can have active in a single clock cycle, as that determines the performance of the system [26]. Additionally, they have to consider how data flows between the hardware components and how resources are distributed throughout the system.

A FPGA only has a certain amount of available space. This space contains all of the configurable logic gates and configurable data paths that can be used to specify a system and support reconfiguration. High performance designs make effective use of the available space on a chip to carry out tasks in parallel. The space constraint is one new limit that is imposed on a model-based approach to hardware that is not present in software. Being able to fit generated designs on chips without wasting space is important for the model-based approach to be successful.

Timing and clock cycles are very important to the performance of hardware systems. Ideally, each component in a hardware device is in use as often as possible to process the most data and achieve the best performance. When working at a higher abstraction level, it is difficult to specify the timing of individual system components while keeping the abstractions intact. Instead, timing decisions will have to be moved to

a lower abstraction layer. The decisions will be made automatically based on sets of rules that may or may not provide optimal solutions.

The space and timing problems are each issues of tradeoffs for a model-based approach. When considering space, or chip area, it is almost certain that a developer manually building a design can find a more optimal method of placing components on a chip than a generation facility. Additionally, because the module is generalized to be reusable in many different situations, it has the additional space overhead of any glue or wrapper logic to its interfaces. A manual design could bypass such generalizations and directly connect components together. The tradeoff here is that while the manual design may make more effective use of space and perform better, the model-based approach is significantly more reusable for a wide variety of circumstances.

When considering timing and clock speeds of components, it is reasonable to assume that a manual design could find a more optimized way than a basic compiler to have the fastest clock speeds for components. The designer can also reasonably achieve a higher degree of parallelism since they can look at the system as a whole and find places that can be tweaked for improvements. A model-based approach would attempt to use coarse-grained timing throughout the system. Instead of allowing any component to run at any clock speed, coarse-grained timing puts parts of the whole system on the same timing scheme. The tradeoff between the manual and model-based approaches is performance for ease of design. The model-based approach sacrifices possible performance gains in order to make the process of designing the system faster and simpler.

How well these tradeoffs are made will determine the effectiveness of the model-based approach in retaining the performance benefits of hardware applications. In software, compilers were initially simplistic and low level programmers could beat them at optimizations. However, as the technology continued to improve and the constraints continued to decrease, compilers took hold and are now a staple of software development. Hardware design is now in the same position that software design was in the past. Similar opportunities are present for improving the ability of developers to create applications and making reconfigurable development more widely available.

4.4 Feasibility of the Model-Based Approach

Given the unique challenges present in modeling hardware, it is important to address whether or not a complete model-based approach is feasible. The tradeoff for raising abstractions is giving up some of the customizations and optimizations that come from manual design. Part of the reason for using a hardware platform for computation is to be able to achieve significant performance gains by specifying the exact hardware layout in a customized architecture. In order to become widespread, the model-based approach needs to address the concern of being able to feasibly develop useful systems on hardware.

As detailed in earlier chapters, a large productivity gap exists between the capacity of today's reconfigurable technology and the ability of engineers to utilize that power. The technology continues to improve and researchers are looking for ways to improve design methods to keep up. In order to increase productivity, the amount of reuse in reconfigurable design must increase. Successfully increasing reuse in software has been strongly tied to raising abstractions [55]. Software abstractions allow programmers to write less code to get the same amount of work done. Less code usually means less work and less time to accomplish a development task. Creating elegant, reusable, or portable code does take additional time per line of code, but the benefits can be reaped later through successful reuse. Because of this, more development time is spent early on in MBE, but reuse, and therefore productivity, increases as times goes on and more components are developed.

Productivity is not solely rooted in the development process of systems. Both development productivity and maintenance productivity are important in considering the amount a programmer can accomplish with the tools at hand [16]. Using higher level languages in software speeds up both development time and maintenance time by reducing the amount of code a developer has to write, test, and maintain. Similarly, in RC, using C-mappers has shown incremental improvements. However, the more dramatic improvements have been seen with moving to generation systems for software production. While a language abstraction can help a programmer solve a problem easier, a generation facility may eliminate the need for the programmer to worry about that aspect of the problem at all. Additionally, lower level functionality is abstracted and

reused throughout the development process. Lower level reusable blocks can be trusted because they have been developed, tested, and standardized. Utilizing abstractions can thus increase both the quality of a system as well as the speed at which it can be developed.

To help developers make good use of higher abstractions, they need good development environments to support them. Ideally, a development environment provides access to a wide assortment of tools that can be used independent of any particular platform. Vendor lock-in is a great situation for a vendor, but terrible for encouraging diversity and collaboration in the reconfigurable computing community. An IDE must enhance reuse and provide the developer with high level views of applications. Being flexible and able to incorporate new technologies is also a key feature with the fast-paced research going on in RC. Common software development enablers like standard interfaces, architectures, and development infrastructure are helpful in increasing productivity for software. It is reasonable to expect an RC development environment to be able to include these features as the field matures, enabling productivity gains for RC, too.

One of the biggest barriers to using a model-based environment is populating the model repository. At the beginning, there is little productivity gain over standard methods while the library of components and modules is being filled. However, as more and more components become available, developers will be able to quickly compose systems and configure them to their needs. As support for model-based development grows and the resources available to developers increase, the impact of the approach will be even greater.

It is difficult to determine whether the repository can ever be considered complete. A number of lower level base components may be created to build larger blocks out of, as seen in computer assisted design tools. At a higher abstraction level, domain components will be put in the repository to be reused in other projects within the same domain. Ideally, a large number of abstract domain components will be made available in the repository and regularly reused to increase productivity. These components will represent common patterns and solutions to problems in that specific domain area. At some point, these components may become old or obsolete, and need to

be replaced or updated in order to maintain confidence in the repository and performance of the components.

The majority of development costs when using a model-based approach are front-loaded. Few components related to a domain area will initially be available at the start of a new project when MBE is first adopted. As more and more components are added, the repository for a given domain is filled out, enabling greater reuse on future projects. MBE lets developers trade more work early in the development cycle for significant reusability later on. If a large community collaborates and contributes, the potential for higher reuse, and therefore productivity, increases even more.

A need for increased productivity in reconfigurable computing design exists. Similarly to how abstractions increased the productivity and availability of software programming, higher level abstractions are the answer to the needs of reconfigurable development. Model-based engineering goes a long way towards supporting the use of higher level abstraction for system design and can give hardware developers a jump start on catching up with the productivity gap.

4.5 Future Enhancements for the Modeling Environment

The MDRC framework presented in Chapter 3 is a prototype that covers some of the basic functions of a model-based approach to RC design. The features of a platform independent modeling environment, component repository, and generation facility to create the platform specific model are the major requirements for a functioning model-based approach. The prototype is a good start, but there is still much room for improvement.

Validations are an obvious feature that should be included before the MDRC environment is available to actual users as more than a research tool. The CHREC team came up with several different scenarios where different kinds of validations would be useful. Having user-defined validations on user-defined properties is one straightforward area where error-checking can be applied. For example, it does not make sense for a queue to have a negative size. The modeling environment should be able to detect such input errors and display appropriate error messages to the user. Constraint languages,

like Object Constraint Language (OCL), can be used to achieve these types of validations. OCL is already popular with UML and is used in many of Object Managament Group's standards. There is also an opportunity to introduce formal methods into the design process for improved verification.

Connection validations are another useful validation type that can benefit users. By expanding the description of interfaces to include the type of inputs that are accepted, error-checking can be done when interfaces are connected to each other. If a user attempts to connect the output of one interface to the input of another, but those interfaces expect different data to be flowing between them, then the system should flag an error. This check helps to ensure the logical correctness of the PIM and makes generation an easier task at the lower levels.

One of the most desired features for the CHREC team is the ability to work with partial reconfiguration. Being able to reconfigure a device at run-time is one of the major advantages of FPGAs. Having the environment support partial reconfiguration would increase what users can do with it. Currently, there is a way to display a reconfigurable section within a diagram of the modeling environment. The semantics for generation are still undecided at this time. The easiest way to do reconfiguration that the environment could support without much additional work would be total re-rendering of the FPGA. Two different configurations could be stored and swapped out as needed. Partial and dynamic run-time reconfiguration would add even more functionality and flexibility to the environment, but additional research is needed. This would be a great area to integrate other research on describing partially reconfigurable systems.

Improving the usage of connectors is another future goal for the environment. In xADL, a connector is supposed to carry any semantics that occur on a connection. No connectors were used in the demonstration, though there is the opportunity for them. It is difficult, at first, to draw a separation between connectors and components in hardware, since everything ends up as a hardware component. Ideally, connectors would be used when conversion needs to be done between the interfaces of two components, to enable parallelization, or adding other semantics to a connection. In the demonstration, the FIFO queue can be thought of as a connector between the signal source and the modulator. The queue controls the connection between those two components.

Representing the serial to parallel component as a connector may also be possible. That component prepares the signal to be used in the constellation mapping, which can be thought of as doing conversion before the connection reaches the next component.

The repository's functionality can be improved by separating the components into different abstraction layers or domain areas. Some components will inherently be at a higher level of abstraction than other components. Different domain areas have specialized components optimized for use in that area. Making these relations clear to the user is important in providing a clean interface for reuse.

The model-based approach provides a great opportunity to integrate a wide selection of resources and tools. Making the research that the whole RC community is doing available to developers is a big step towards increasing productivity. The success of model-based engineering is only possible through contributions of other developers. An integrated model-based development environment gives developers a platform independent workspace that is based on the strong foundations of research in hardware design.

Currently, the MDRC framework uses an assembly approach to development. If there is not a component in the repository for what the user wants to do, then it is up to them to develop it. The ideal model-based approach has a complex set of transformations that can be used to generate the platform specific implementation directly from the PIM, without needing a previously assembled module. In essence, there needs to be richer semantics in the transforms to parameterize the resulting components for higher degrees of specialization. Understanding the transforms to the degree necessary to implement this level of automation only comes as the field matures, the component repository is populated, and patterns are discovered. A considerable research effort will be necessary to develop transformations that can generically generate optimized hardware modules through this method. A transformational approach has tremendous potential for changing the way developers work with hardware design, but it is outside the scope of this thesis.

Chapter 5
Conclusions

Reconfigurable computing is currently going through many of the same problems encountered by software engineering in the past. The power and potential of the technology continues to increase while the capabilities of engineers and developers lag behind. This productivity gap is caused by a lack of the necessary abstractions with which to reason about systems at a high level and properly reuse low level components effectively. Reconfigurable development presents its own unique challenges related to developing applications for hardware on a different architecture from typical software engineering.

Model-based engineering *can* help solve the productivity gap by raising the abstraction levels for developers and enabling reuse. A model-based approach provides a separation of concerns between the application logic and implementation of a system. MDA, a software variant of MBE, has been successfully used on software projects commercially and in research to improve programmer productivity. The benefits from MBE in the areas of reusability, portability, automation, and traceability are very desirable for hardware development. When many projects will be carried out in the same domain area, there is a great opportunity for MBE to be used to produce productivity benefits. As the domain is explored and components are added to the repository, it becomes easier to reuse old work products in newer projects and reusable patterns surface that can be carried over to other systems. Long-term development projects can also benefit from MBE, as the application logic remains portable and more is available for reuse later in the life cycle. MBE would not be as beneficial for one-time tasks or projects where reuse will not be possible. This research explored the questions posed in Section 1.3 centered on using MBE for reconfigurable system development.

Is using MBE for FPGA development feasible? This thesis presented a model-driven framework called MDRC for developing RC applications on FPGAs. MDRC

provides a platform independent modeling environment based on a flexible metamodel. The environment gives users the ability to define, store, reuse, connect, and fully customize components. The framework also contains a facility for generating the low level code that the application needs to run on hardware. The CHREC team demonstrated the feasibility of MBE by implementing a transmitter for a Xilinx FPGA. The development process involved creating the modules, making them available in the repository, building the platform independent description of the transmitter, and generating the HDL modules for the correct architecture. As the environment and repository become more mature, the number of situations where MBE can feasibly be applied increases.

Can reuse and portability be supported? The MDRC framework supports both component reuse and portability between platforms. Reuse is achieved through a repository where components can be stored and monitored through version control. Users of the modeling environment can use stored components and build new components from others to store and reuse later. Because of the platform independence of the modeling environment, portability is achieved. The implementation can be generated for any desired platform, as long as the components being used are available for that platform. When necessary, a developer can add new components to the repository and rules for how to generate the underlying modules for specific architectures. The degree to which the repository has matured has a strong effect on how much reuse and portability users can get out of the MBE approach. A full repository that has many variations of components for different architectures would be an extremely valuable asset for future development projects.

Are productivity gains possible by using a model-based development environment? Based on the success of the scenario and the potential of the modeling environment, the researchers are convinced that a model-based approach to reconfigurable development can provide significant productivity improvements. Through reuse and automation, developers can build larger hardware applications with greater complexity faster than would be possible without the abstractions provided by the model-based approach. The model-based development environment allows users to assemble components to design hardware applications at a high level of abstraction. The potential

for reuse, portability, and automation that comes from the higher level abstractions can have an enormous impact on productivity. The population of the repository is a strong determinant of how much productivity can be gained. Little to no productivity is gained when the repository is empty, but as it increases, so does the ability of developers to reuse old work products to quickly develop systems.

How do these compare with similar gains/losses in software engineering? Productivity gains in software have come from abstraction, reuse, process improvements, and automation. Each of these elements plays a role in RC system development, and specifically with MBE. Large scale RC applications cannot reasonably be developed with low-level hardware languages, just like the numerous software systems of today could not have reasonably been developed in assembly. Similarly to software, RC developers must sacrifice some performance optimizations at run-time to improve their performance at design-time. MBE aims to achieve a high degree of reuse, and thus productivity, through high-level abstractions and automation. Software model-based approaches perform better as their domain repository becomes more complete and mature; the same should be true for RC.

Model-based engineering has proven to be a powerful force in software development for raising abstractions and enhancing the capabilities of developers. A model-based approach to reconfigurable hardware design can bring about new levels of reuse and integration that will impact developer productivity. An integrated development environment that supports model-based engineering principles can help bridge the productivity gap facing the field of reconfigurable computing

5.1 Contributions

This research was carried out as part of an initiative with the Center for High-Performance Reconfigurable Computing (CHREC) at Virginia Tech. The CHREC team consisted of several members from both the Computer Science department and the Electrical and Computer Engineering department. The ECE members served as the FPGA experts, while I helped to integrate model-based engineering into the design process.

My contributions included the development of the Eclipse-based modeling environment, the customization of the xADL instance schema to describe our reconfigurable architecture, and the creation of the metamodel for representing RC components and systems. The two biggest challenges throughout the two semesters I was involved with the project were the development of the modeling environment to support MBE principles and helping the hardware specialists to think and work in a model-based way. The main challenge with the environment involved the creation of the metamodel to support reconfigurable hardware development and providing the graphical representation to work with that hardware system. Changing the way the hardware designers thought about the design process was also challenging. Most hardware designs are highly integrated to minimize space and take advantage of optimizations. Breaking out the system components into reusable modules was initially difficult, but provided the opportunities for increasing reuse and productivity.

The ECE members were heavily relied upon to create the hardware system designs and the individual hardware modules. They helped develop the ASM, using their skills to decide the best way to generate the HDL modules for the high-level components. They served as the CHREC team's experts in software-defined radio.

References

[1] Benkrid, Khaled. "High Performance Reconfigurable Computing: From Applications to Hardware." IAENG International Journal of Computer Science, vol. 35, issue 1. February, 2008.

[2] Bohner, S., R. Ravichandar, et al. (2007). "Model-based engineering for change-tolerant systems." Innovations in systems and software engineering. **3**(4): 237.

[3] P. Sedcole. *Reconfigurable Platform-Based Design in FPGAs for Video Image Processing*. PhD thesis, Imperial College London, January 2006.

[4] Hauck, S. The Future of Reconfigurable Systems. Keynote Address, 5th Canadian Conference on Field Programmable Devices . 1998. Montreal.

[5] Katherine, C. and H. Scott (2002). "Reconfigurable computing: a survey of systems and software." ACM Comput. Surv. **34**(2): 171-210.

[6] Todman T. J. Todman, S. J. E.Wilton, O. Mencer,W. Luk, G. A. Constantinides, and P. Y. K. Cheung. Reconfigurable Computing: Architectures and Design Methods. In *IEE '05: IEE Proceedings Computers and Digital Techniques*, volume 152, pages 193–207, 2005.

[7] Walid A. Najjar, Wim Bohm, Bruce A. Draper, Jeff Hammes, Robert Rinker, J. Ross Beveridge, Monica Chawathe, Charles Ross, "High-Level Language Abstraction for Reconfigurable Computing," *Computer*, vol. 36, no. 8, pp. 63-69, Aug., 2003.

[8] Rajopadhye, S., Gupta, G., and Renganarayana, L. 2008. A domain specific interconnect for reconfigurable computing. In *Proceedings of the 2008 ACM SIGPLAN-SIGBED Conference on Languages, Compilers, and Tools For Embedded Systems* (Tucson, AZ, USA, June 12 - 13, 2008). LCTES '08. ACM, New York, NY, 79-88.

[9] Z. Guo, D. C. Suresh, W. A. Najjar. Programmability and Efficiency in Reconfigurable Computer Systems, Workshop on Software Support for Reconfigurable Systems, held in conjunction with the Int. Conf. Of High-Performance Computer Architecture (HPCA), Anaheim, CA, February 2003.

[10] Mellor, S. J. (2004). MDA Distilled: Principles of Model-Driven Architecture. Addison Wesley.

[11] Reiner, H. (2004). The digital divide of computing. Proceedings of the 1st conference on Computing frontiers. Ischia, Italy, ACM.

[12] Object Management Group (2006). "Technical Guide to Model Driven Architecture: The MDA Guide v1.0.1" http://www.omg.org/mda/presentations.htm. Object Management Group, Inc.

[13] Sendall, Shane, Jochen Küster, 2004, Vancouver, Canada, Taming Model Round-Trip Engineering, OOPLSA´04.

[14] Selic, B. (2003). "The pragmatics of model-driven development." IEEE software **20**(6): 19.

[15] Pinto, A., Bonivento, A., Sangiovanni-Vincentelli, A. L., Passerone, R., and Sgroi, M. 2004. System level design paradigms: Platform-based design and communication synthesis. In *Proceedings of the 41st Annual Conference on Design Automation* (San Diego, CA, USA, June 07 - 11, 2004). DAC '04. ACM, New York, NY, 537-563.

[16] C. Stevenson. *Software Engineering Productivity - A Practical Guide*. Chapman & Hall, 1995.

[17] Kleppe, A., J. Warmer, et al. (2003). MDA Explained, The Model Driven Architecture: Practice and Promise. Addison Wesley.

[18] Fong, C. K. (2007). Successful Implementation of Model Driven Architecture: A Case Study of How Borland Together MDA Technologies Were Successfully Implemented in a Large Commercial Bank. Borland White Paper. http://www.borland.com/resources/en/pdf/products/together/together-successful-implementation-mda.pdf

[19] Jean Bezivin, Slimane Hammoudi, Denivaldo Lopes, Jouault Jouault, "Applying MDA Approach for Web Service Platform," *edoc*, pp. 58-70, Enterprise Distributed Object Computing Conference, Eighth IEEE International (EDOC'04), 2004.

[20] Van Gorp, P.; Janssens, D.; Gardner, T., "Write once, deploy N: a performance oriented MDA case study," *Enterprise Distributed Object Computing Conference, 2004. EDOC 2004. Proceedings. Eighth IEEE International* , vol., no., pp. 123-134, 20-24 Sept. 2004.

[21] Cougaar architecture document (2004) Version for Cougaar 11. 4. 2004, BBN Technologies: Technical report, p 74. http://www.cougaar.org/

[22] Bohner S, George B, Gracanin D, Hinchey M (2005) Formalism challenges of the Cougaar model driven architecture. Lecture notes in computer science, Springer, Heidelberg. GmbH (selected and expanded from formal approaches to agent-based systems—FAABS III)

[23] Xilinx, Inc. Xilinx Core Generator product page (2008). (Last accessed 07 November 2008). http://www.xilinx.com/products/design_tools/logic_design/design_entry/coregenerator.htm

[24] National Instruments, LabVIEW home page. (Last accessed 14 November 2008) http://www.ni.com/labview/

[25] Backus, J. 1978. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM 21*, 8, (August), 613-641.

[26] Bondalapati, K., and Prasanna, V.K.: 'Reconfigurable computing systems', Proc. IEEE, 2002, 90, (7), pp. 1201–1217

[27] A.Cosoroaba, F. Rivoallon, "Achieving Higher System Performance with the Virtex-5 Family of FPGAs." http://www.xilinx.com/support/documentation/white_papers/wp245.pdf

[28] Altera Corporation. White Paper: FPGA Architecture. http://www.altera.com/literature/wp/wp-01003.pdf

[29] G. Goslin, "A guide to using field programmable gate arrays (FPGAs) for application-specific digital signal processing performance," Tech.Rep., Xilinx Inc., San Jose, 1995.

[30] Damaševičius, R., Majauskas, G., and Štuikys, V. 2003. Application of design patterns for hardware design. In *Proceedings of the 40th Conference on Design Automation* (Anaheim, CA, USA, June 02 - 06, 2003). DAC '03. ACM, New York, NY, 48-53.

[31] M. Gokhale, J.M. Stone, J. Arnold and M. Kalinowski, "Stream-oriented FPGA computing in the Streams-C high level language", Proc. Symp. On Field-Programmable Custom Computing Machines, IEEE Computer Society Press, 2000.

[32] P.A. Jackson, B.L. Hutchings and J.L. Tripp, "Simulation and synthesis of CSP-based interprocess communication", Proc. Symp. on Field-Programmable Custom Computing Machines, IEEE Computer Society Press, 2003.

[33] S. Gupta, N.D. Dutt, R.K. Gupta and A. Nicolau, "SPARK: a high-level synthesis framework for applying parallelizing compiler transformations, Proc. International Conference on VLSI Design, January 2003.

[34] M. Weinhardt and W. Luk, "Pipeline vectorization, IEEE Trans. on Computer-Aided Design, Vol. 20, No. 2, 2001.

[35] O. Mencer, D.J. Pearce, L.W. Howes and W. Luk, “Design space exploration with A Stream Compiler”, Proc. Int. Conf. on Field Programmable Technology, IEEE, 2003,

[36] Celoxica, Handel-C Language Reference Manual for DK2.0, Document RM-1003-4.0, 2003.

[37] J.G. De Figueiredo Coutinho and W. Luk, “Source-directed transformations for hardware compilation”, Proc. Int. Conf. on Field-Programmable Technology, IEEE, 2003.

[38] A. Yamada, K. Nishida, R. Sakurai, A. Kay, T. Nomura and T. Kambe, “Hardware synthesis with the Bach system”, Proc. ISCAS, IEEE, 1999.

[39] Xilinx, Inc. Xilinx Logic Design Products. (Last accessed 07 November 2008) http://www.xilinx.com/ise/logic_design_prod/index.htm

[40] Altera Corporation. Altera Products. (Last accessed 07 November 2008) http://www.altera.com/products/prd-index.html

[41] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[42] McMillan, S. and Guccione, S. 2000. Partial Run-Time Reconfiguration Using JRTR. In *Proceedings of the the Roadmap To Reconfigurable Computing, 10th international Workshop on Field-Programmable Logic and Applications* (August 27 - 30, 2000). R. W. Hartenstein and H. Grünbacher, Eds. Lecture Notes In Computer Science, vol. 1896. Springer-Verlag, London, 352-360.

[43] Leijten-Nowak, K.: Template-Based Embedded Reconfigurable Computing. Ph.D. thesis, Technische Universiteit Eindhoven (2004)

[44] A. Sangiovanni-Vincentelli, and G. Martin. A Vision for Embedded Systems: Platform-Based Design and Software Methodology. *IEEE Design and Test of Computers*, Vol. 18, No. 6, 2001, pp. 23-33.

[45] Alberto Sangiovanni-Vincentelli. "Defining Platform-based Design". *EEDesign of EETimes*, February 2002.

[46] Eric M. Dashofy, André van der Hoek, and Richard N. Taylor
In *Proceedings of the 24th International Conference on Software Engineering (ICSE2002)*, Orlando, Florida.

[47] Eclipse main website (2008). (Last accessed 07 November 2008). http://www.eclipse.org

[48] Eclipse Graphical Modeling Framework main website (2008). (Last accessed 07 November 2008). http://www.eclipse.org/modeling/gmf/

[49] Subversive main website (2008). (Last accessed 07 November 2008). http://www.eclipse.org/subversive/

[50] xADL main website (2005). (Last accessed 07 November 2008) http://www.isr.uci.edu/projects/xarchuci/

[51] Recio, Adolfo. ASM Diagrams. CHREC wiki homepage (2008). (Last accessed 07 November 2008). http://www.ccm.ece.vt.edu/cgi-bin/twiki/view/CHREC/WebHome

[52] Tuttlebee, W.H.W., "Software-defined radio: facets of a developing technology," *Personal Communications, IEEE [see also IEEE Wireless Communications]* , vol.6, no.2, pp.38-44, Apr 1999

[53] Lehman, M. and J. C. Fern´andez-Ramil (2002). "Software Evolution and Software Evolution Processes." Annals of software engineering **14**(1/4): 275.

[54] DeHon, A. and Wawrzynek, J. 1999. Reconfigurable computing: what, why, and implications for design automation. In *Proceedings of the 36th ACM/IEEE Conference on Design Automation* (New Orleans, Louisiana, United States, June 21 - 25, 1999). M. J. Irwin, Ed. DAC '99. ACM, New York, NY, 610-615.

[55] Krueger, C. W. 1992. Software reuse. *ACM Comput. Surv.* 24, 2 (Jun. 1992), 131-183.