OpenJ: An Extensible System Level Design Language

Jianwen Zhu, Daniel D. Gajski
CECS, Information and Computer Science
University of California
Irvine, CA 92717-3425, USA
jzhu, gajski@ics.uci.edu

Abstract

There is an increasing research interest in system level design languages which can carry designers from specification to implementation of system-on-a-chip. Unfortunately, two of the most important goals in designing such a language, are at odds with each other: Heterogeneity requires components of the system to be captured precisely by domain specific models to simplify analysis and synthesis; simplicity requires a consistent notation to avoid confusion. In this paper, we focus on our effort in resolving this dilemma in an extensible language called OpenJ. In contrast to the conventional monolithic languages, OpenJ has a layered structure consisting of the kernel layer, which is essentially an object oriented language designed to be simple, modular and polymorphic; and the open layer, which exports parameterizable language constructs; and the domain layer which precisely captures the computational models essential for embedded systems. The domain layer can be provided by vendors via a common protocol defined by open layer which enables the supersetting or/and subsetting of the kernel. A compiler has been built for this language and experiments are conducted for popular models such as synchronous, discrete event and dataflow.

1 Introduction

Advances in VLSI technology have made it possible for system on a chip. The design of such systems imposes new challenges to design automation systems. Most of the challenges stem from the complex nature of the system functionality, implementation technology and design process. At any design stage, the system can be best abstracted as a set of interacting components, each of which behaves according to certain *computational models*. Many useful models, whose semantics are often captured by dedicated languages, have been developed [1].

It is not surprising that the current practice of codesign methodology uses different languages for system specification, called *co-specification*, and uses even more languages when the design process proceeds. However, users of this methodology tend to spend more time on discovering the differences of these languages and making the corresponding tools work together, rather than the problems themselves. It is thus desirable to have one common language for the specification, synthesis and validation of the entire system to cover the entire design process, or cover as much as possible. Recent IEEE effort on System Level Design Language (SLDL) standardization [2], represents this trend. While an attractive idea since no existing languages are designed for this purpose, this task is non-trivial. One fundamental barrier lies between the *simplicity* requirement for any language to be successful, and the *heterogeneity* requirement which insists all useful models to be captured.

In this paper, we demonstrate our effort in resolving this dilemma by a language called OpenJ . This paper makes several contributions. First, analogous to the microkernel architecture in operating systems, we propose a layered language architecture, in contrasts to the monolithic architecture in conventional languages. This architecture allows sublanguages which precisely capture the computational models, called the domain layer, to be easily built by vendors on a common language substrate, called the kernel layer, via the open layer, where parameterized constructs are exported. Second, although the kernel layer has been heavily influenced, and is in fact derived from Java [3], we have made significant changes which lead to a more powerful type system and simpler runtime system. Third, we have defined a protocol between the kernel compiler and the domain compiler, in order to efficiently define domain languages by supersetting and/or subsetting the kernel language.

The rest of the paper is organized as follows. Section 2 categorizes related works. Section 3 discusses in more detail about the architecture. Section 4 describes the kernel, where its differences with Java are emphasized. Section 5 described the open layer and the protocol. Section 6 described the implementation and our experiment with various popular computational models.

2 Related Works

Related works fall into the following categories.

The *library extension* approach leverages the expressive power of general purpose object oriented languages. in [5] [6] [7], C++ class libraries are developed to capture the hardware semantics. In [8], extensive analysis is performed on Java specification to discover the task/loop level concurrency. In [9], a Java class library is developed to capture the synchronous reactive semantics, and a “policy of use” is imposed upon the Java specification. While hardware and software can be specified in the same language, this methodology suffers two drawbacks. First, in contrasts to the variety of constructs that can be introduced in our language to capture new semantics, the only mechanism available in this approach is via class library. Even with the disguise of operator overloading, expressiveness of library calls is limited. In addition, it is not clear whether the library code is for simulation purpose or for synthesis purpose. Second, without the semantic checkers available in our domain languages, a policy of use, suggested in [9], has to be imposed in the form of design styles, which is well known to be unreliable.

The *language extension* approach extends existing languages. For example, SpecC [10] extends ANSI C with features that help to describe embedded systems.

The *new language with homogeneous model* approach makes no attempt to be compatible with any existing languages, instead, a new language is crafted based on certain computational model. V++ [13], a new synthesizable hardware description language, is based on the synchronous model, but with better system level support by solving the composition problem inherent in traditional synchronous languages such as Esterel [11] and LUSTRE [12].

OpenJ is a new language with heterogeneity support. Its kernel language can be used as behavioral description that can be compiled into either assembly or RTL hardware. Its domain layer contains domain languages that can describe popular models such as discrete event, dataflow and synchronous models. It also serves as a foundation to experiment with new models.

Programming language pioneers have suggested that new languages should not incorporate "unexperimented" features. Fortunately, our basic approach towards resolving the heterogeneity/simplicity conflict finds cousins targeting different problems in different contexts, which helps to illustrate the value of our approach: The need for domain specific languages is well known and a USENIX conference has devoted to the subject. The layered language architecture can also be found in the Rapide language [14], although Rapide is not extensible. The language extensibility is also allowed in the hardware description language CONLAN [15], although it involves a more complex mechanism and its kernel is not a full fledged object oriented language.

3 Language Architecture

Unlike traditional programming languages, OpenJ has a layered architecture . At the bottom of the architecture lies the kernel layer, which is essentially a pure object oriented language. The constructs defined in the kernel layer are classes, fields, methods, types, variables, statements and expressions. In the middle is the open layer, which contains the set of constructs whose keywords are defined by the top layer, called the domain layer. Domain layer contains a set of well defined languages which exactly capture certain computational models, which are either specification models, for example, the dataflow model to represent signal processing systems; or implementation models, for example, the discrete event model to represent gate level netlist. Correspondingly, the OpenJ compiler contains the kernel compiler as well as a set of domain compilers, which interact via a common protocol called the domain protocol.

This architecture meets well with the heterogeneity requirement. The existence of the domain layer allows the definition of domain specific languages which can precisely capture the system component behaviors governed by particular computational models. Such preciseness translates to the efficiency of synthesis and simulation tools that can take advantage of the domain knowledge, which is often difficult, if not impossible, to infer from more general models. For example, given the partial order explicitly captured by the data flow model, the domain compiler can compute an optimal schedule which minimizes memory overhead, while meeting the performance constraints. Under the assumption that all behavior execution under the synchronous model is aligned with a common clock, the simulation can bypass the event management that are unavoidable in the more general discrete event model, and hence can be significantly faster.

This architecture meets the simplicity requirement better than monolithic languages and the multi-lingual approach. Releaved from the burden of capturing domain specific models, the kernel layer can be kept very simple. In addition, domain specific languages can share constructs in the kernel that are universal, such as those contribute to the modularity, parameterizability, and type safety of the program. For example, instead of extending hardware description languages with the abstract data type mechanisms that prevail software languages (for example, the IEEE OO-VHDL standardization), hardware domains in OpenJ can immediately use the one provided by the kernel "for free". Furthermore, domain languages are *closed* in the sense that constructs defined in one domain never interact with constructs defined in another domain, whereas in the monolithic languages, for example, the meaning of the combination of any statements has to be defined. The language architecture chosen, as well as the language definition itself, leads to the simplicity of OpenJ , which directly translates to the ease of learning from the user's part, and ease of compiler construction from the vendor's part.

4 Kernel Layer

As its name suggests, OpenJ kernel is derived from Java. Our favor of Java's design philosophy over C++'s is not accidental. Among the most important are: First, OpenJ kernel is intended to be a *new* language, and hence do not have to be compatible at the language level to any other languages such as C. Many redundant constructs, for example, functions and static member functions, struct and class in C++, can thus be avoided. Second, OpenJ kernel is intended to be a *pure* object oriented language, in other words, every variable in the program is an object, and an object is accessed either by value or by reference consistently, but not both. This leads to conceptually simpler programs. Third, OpenJ kernel is intended to be a *strongly typed* language, in other words, arbitrary type cast in C++ is not allowed. This leads to safer programming and much simpler alias analysis in the synthesis tools. Forth, OpenJ kernel is intended to be *modular*, in other words, the compiler maintains a set of packages composed of separately compiled modules, instead of the cumbersome preprocessing mechanism of C++. More discussions can be found in [16].

However, OpenJ kernel is neither superset nor subset of Java. Instead, to be more elegant, powerful and convenient, it has been redesigned based on (a) results in modern functional and object oriented language research [4]; (b) lessons learned in software programming and compilation for embedded processors [17]; (c) the need for system (including hardware) modeling.

4.1 Syntax

The program written in OpenJ kernel is organized into a set of *type* specifications, or *classes*. Each class is contained in a *package*, which defines a *name space*. A class contains a set of *fields*, which helps to model the runtime state of the program; a set of *methods*, which represent the functions that modifies the program state; and a set of nested classes. In addition, a class in OpenJ also serves as the basic unit of *encapsulation*. Every unique type, for example, a nested type specified by a nested class, introduces a new name space.

The behavior of a normal method is specified by a set of *statements*, representing control flows over sets of *expressions*, which in turn are trees composed of constants, local accesses, field accesses, object allocations/deallocations, *method dispatches*, and syntactic sugars that can finally be reduced to method dispatches. A method dispatch can be static, where the behavior of the method is determined at compile time; or dynamic, where the behavior of the method is determined at runtime.The behavior of a method can be left unspecified, if the method is either *primitive*, or *native*, or *abstract*. The behavior of a primitive method is determined at compile time, for example, directly mapped to machine instructions. The behavior of a native method is resolved at link time and hence can be specified by other languages such as C. An abstract method implies an entry in the dispatch table of the associated type, which helps to resolve the behavior at runtime.

An allocation expression or a method dispatch expression is associated with its *base*, which is either a type or an object. For example, `int` in `int.add( a, b )` is a type. On the other hand, `adder` in `adder.add( a, b )` is an object. The conventional operators are defined as syntactical sugars for method dispatches. For example, the expression `a +[int] b` is equivalent to `int.add( a, b );` and the expression `a +[adder] b` is equivalent to `adder.add( a, b )`. If the base is omitted, the default base is the type of the first argument. For example, if `a` is of `int` type, then the expression `a + b` is equivalent to `int.add(a,b)`. Note that here the same effect of C++ *operator overloading*, a desired feature not present in Java, is achieved. However, unlike C++, the overloaded operator in OpenJ is not tied to the type of its arguments, instead, it is tied to its base, which can be specified.

The fields and methods of a type can be *inherited* via *implementation inheritance*. The dispatch table of a type can be inherited by *interface inheritance*.

4.2 Type System

Three categories of types are distinguished in OpenJ , each of which ensures a different discipline on the access of the objects with this type. An object of *reference type* is always accessed by reference, and associated methods are always staticly dispatched. An object of *interface type* is also accessed by reference, but the associated methods are always dynamically dispatched. An object of *value type* is always accessed by value, and statically dispatched. For convenience, the *enumeration* type and the *tuple* type are provided as special cases of value type. Note that tuple types can be used to return multiple values for a method, which is impossible in Java without dynamic allocation of objects. The assignment expression of tuple type is interpretated as parallel assignment, which is also convenient to model RTL operations. For example, the expression `(R1, R2) = (R2, R1)` effectively swap the content of `R1` and `R2`.

A transitive *subtype* relation is established via implementation inheritance (indicated by a `extends` clause) and interface inheritance (indicated by a `implements` clause). An expression of certain type can appear any place where expression of its super type is expected.

A type specification in OpenJ can be parameterized by other types. Constraints on the type parameter can be specified by the `extends` and `implements` clauses. Such *bounded parametric polymorphism* ensures that type checking can be performed at compile time. This is in contrasts to the template mechanism of C++, where full type checking has to be delayed until a concrete type is instantiated.

The type system has been made more *powerful* than that of Java thanks to the adoption of parametrized types and value types. It is also made more convenient thanks to the two special cases of the value type: the enumeration type, which is important to model checking applications which require the program state to be finite; and the tuple type, which can be considered as value type "on demand". OpenJ 's type system is also more *elegant* than that of Java, since there are no "exceptions to the rules" in the type system. For example, the primitive types and array types of Java have different behavior than a normal type. In OpenJ , such "first class citizens" simply does not exist. An example helps to clarify the importance of such elegance. Suppose a program has to be developed for an application specific processor which contains a datapath operating on 24 bit integer values, and other than normal arithmetic operations, it also contains the "irregular" operations such as saturation add. If Java or C/C++ is used, since there is no primitive types of 24-bit integer, a larger data type has to be used. While code can be developed for the bit-true simulation, it is difficult for the compiler to recognize the simulation code to be the normal arithmetic instructions or the idioms such as saturation add provided by the processor. Alternatively, the ad hoc "DSP extension" of the language or assembly level programming has to be performed [17]. In sharp contrasts, the value class and the primitive method of OpenJ can help to directly exploit machine resources without changing the language.

4.3 Runtime System

In favor of a slim runtime, OpenJ elects to drop the garbage collection memory model adopted by Java. Instead, user can explicitly select the memory model via the base of the allocation expression. In the form of types or objects, the memory models, or the memory managers, can be either predefined by the runtime system, or customized by implementing a standard memory manager interface. For example, the memory manager of the expression `new Test()` is defaulted to the type `heap`. The compiler aware manager `auto` in the expression `new [auto] Test()` allocates an object from the stack and deallocates it when the scope of the expression is exited. In the form of an object, the memory manager `arena` in the expression `new [arena] Test1()` and `new [arena] Test2()` allocates two objects from the stack of memory it maintains respectively and deallocate them in batch. Note that the behavior of the memory managers can be easily synthesized into reasonable sized embedded software or ASIC, without worrying about the 10k-line garbage collector.

We dropped Java's "built-in" support from concurrency and synchronization, since it is unfortunately neither general nor efficient enough for our purposes. Instead, in the runtime library, a set of low level primitives are exported which only allows the management of context switch. It is up to the domain languages to define the concurrency models of their own. And the corresponding domain compiler might generate code which safely access the primitives provided. Driven by the same requirement for simplicity, The exception handling mechanism is also left to the domain layer.

The runtime system of OpenJ , appeared as the package `j.lang`, hence becomes extremely small. In fact, the majority of the types defined in this package are those value types with primitive methods, such as *int, long, float, double* etc. The runtime overhead associated with object oriented programming, such as runtime type information and method dispatch table, can also be selectively suppressed by the compiler, if they are not used at all.

5 Open Layer and Domain Protocol

The open layer of OpenJ defines a set of constructs, called the open constructs, whose parameterized keywords are defined by the domain languages. The following constructs are provided:

- Every type, field and method declaration can have a list of clauses, each of which can be a list of types or expressions. An example of type clause can be `throws Exceptions1, Exception2` used for Java style exception handling. An example of expression clause can be the Eiffel style assertion facility such as the `invariant` clause associated with a class to specify invariant properties; and the `requires` and `ensures` clause associated with a method to specify preconditions and postconditions. Note that these properties can be not only the specification for model checking, but also the "don't care" conditions to avoid *overspecification*, and hence a great help to the analysis and synthesis tools.
- Every field and method declaration can have a list of modifiers. For example, a modifier `inline` can be attached to a method to indicate the designer's hint to the compiler.
- There are a set of parameterized statements with predefined grammar. An example is the `waitfor` statement in Figure 2.
- There are a set of parameterized expression with predefined grammar. For example, the expression `printf(``%d%f'', i, f)` can be used in OpenJ to conveniently replace its type-unsafe counterpart in C.

Note that here we insist the form of the open constructs to be predefined, and only their keywords vary. This ensures the syntax of the domain languages maintain the same look and feel and is always "familiar" to the user.

The domain languages are defined by the corresponding domain compilers. When the domain name followed the type declaration is encountered by the parser of the kernel compiler, the corresponding domain compiler is activated to perform the full compilation. The kernel compiler and domain compiler interact via a set of abstract interfaces called the domain protocol. The protocol contains several subprotocols, each of which consists of a kernel part, which the kernel compiler has to implement, and a domain part, which the domain compiler has to implement.

The *syntax protocol* defines the grammar of the domain language by registration of the parameterized constructs. This essentially define the domain language by supersetting the kernel. The *semantic protocol* defines the type checking rules of the domain language. This essentially restricts the previously defined language to be the desired one. The *transform protocol* defines how the new constructs are transformed into the kernel constructs.

```
public interface MetaChecker {
    void startCheckClass( MetaSymb c );
    void endCheckClass( MetaSymb c );
    void startCheckMethod( MetaSymb mtd );
    void endCheckMethod( MetaSymb mtd );
    void startCheckStmt( MetaStmt stmt );
    void endCheckStmt( MetaStmt stmt );
    void checkType( MetaSymb ty );
    void checkField( MetaSymb fld );
    void checkVariable( MetaSymb v );
    MetaExpr checkExpr( MetaExpr expr );
    }
```

Figure 1. Domain part of the semantic protocol.

The semantic and transform protocols are designed to be efficient so that they do not require a separate pass in the compilation process. For example, the domain part of the semantic protocol shown in Figure 1 defines a set of abstract methods. The type checkers of both the kernel compiler and the domain compiler implement this same interface. The kernel compiler can then pipe the calls to these functions while walking over the abstract syntax tree.

6 Implementation and Experiment

We followed a very traditional way for the development of OpenJ compiler. A simple compiler is first developed in an existing language, in our case Java, to translate OpenJ program to C++. Using this compiler as the development environment, we then developed the OpenJ runtime as well as a discrete event simulation backplane using the OpenJ language. A full-fledged, retargetable compiler infrastructure is then developed, still in OpenJ itself. Finally, we are able to bootstrap the compiler using its C backend. While a much more expensive approach than developing in a mature language, this procedure does give us first hand experience with the language. In fact, it helped us to change the language specification several times during the course of the development.

To exercise OpenJ 's capability, we built domain compilers for the discrete event domain; the synchronous domain [11], [12]; the synchronous dataflow [1]; and the C domain, which helps to import or export C functions. Figure 2 shows an example written in the discrete event domain. Here, `always` is a method modifier indicating that the method is a process. `waitfor` is the delay statement. `await` is the synchronization statement. Given syntactical extensions such as these, complex type checking is performed by the domain compiler. For example, a method with `always` modifier should take no argument, returns nothing and be timed. A timed method contains only timed statements. A timed statement is either a delay statement, synchronization statement, a block statement containing only timed statements, or conditional statement and loop statement whose body is timed. The body of delay statement and synchronization statement should not be timed. When the program passed all the checking, the domain compiler then transform the new constructs into kernel constructs. For example, the delay statement is transformed into method calls to the simulation backplane. Finally, it is compiled into the executable.

```
public class DiscreteEventExample
        domain devent {
    event    e1, e2;
    always void foo() {
        waitfor( 1 ) { ... }
        await( e1 ) { ... }
        await( e2 ) { ... }
        }
    }
```

Figure 2. Discrete event domain example.

We are using OpenJ in a daily basis mainly for software development. In addition to the compiler, simulator, and runtime system, which themselves represents over 60000 lines of code, we have developed a number of applications to test our domain languages. One example is an ATM transceiver written in discrete event domain. We have also developed a number of DSP applications based on public domain C implementations. One example is a GSM speech codec.

7 Conclusion and Outlook

In conclusion, The layered architecture can lead to a powerful yet simple language suitable for heterogeneous system design. While we believe allowing too many computational models to coexist in one language is equally a bad idea than allowing only one model, the extensibility of OpenJ provides a good foundation for experiment before research is mature enough to decide the exact set of models and their exact semantics.

8 Acknowledgement

The authors would like to thank En-shou Chang for his careful review of the initial manuscript and the anonymous reviewers for providing constructive comments and pointing out the CONLAN effort.

References

[1] E.A. Lee, A. Sangiovanni-Vincentell. *Comparing Models of Computation*. Proceedings of the International Conference on Computer-Aided Design, November, 1996.

[2] S.E. Schulz. *The New System-Level Design Language*. Integrated System Design, July, 1998.

[3] J. Gosling, B. Joy, and G. Steele. The Java Language Specification. Reading, MA: Addison-Wesley, 1996.

[4] M. Odersky, P. Wadler. *Pizza into Java: Translating Theory into Practice* Proceedings of 24th ACM Symposium on Principles of Programming Languages, Paris, France, January, 1997.

[5] J.T. Buck, S. Ha, E.A. Lee, D.G.Messerschmitt. *Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systesms*. International Journal on Computer Simulation, Vol. 4, April, 1994.

[6] S. Liao, S. Tjiang, and R. Gupta. *An Efficient Implementation of Reactivity for Modeling Hardware in the Scenic Design Environment*. Proceeding of 34th DAC, 1997.

[7] P. Schaumont S. Vernalde L. Rijnders M. Engels I. Bolsens. *A Programming Environment for the Design of Complex High Speed ASICs*. Proceeding of 35th DAC, 1998.

[8] R. Helaihel and K. Olukotun. *Java as a Specification Language for Hardware-Software Systems*. Proceedings of the International Conference on Computer-Aided Design, November 1997.

[9] J. S. Young, J. M. MacDonald, M. Shilman, A. Tabbara, P. Hilfinger, A. R. Newton. *Design and Specification of Embedded Systems in Java Using Successive, Formal Refinement*. Proceedings of 35th DAC, June, 1998.

[10] J. Zhu, R. Dömer, D.G. Gajski. *Syntax and Semantics of SpecC Language*. Proceedings on the Seventh Workshop on Synthesis and System Integration of Mixed Technologies, Dec. 1997.

[11] G. Berry and G. Gonthier. *The ESTEREL Synchronous Programming Language: Design, Semantics, Implementation*. Science of Computer Programming, vol. 19, pp.87-152, 1992.

[12] C.N. Halbwachs, et. al. *The Synchronous Data Flow Programming Language LUSTRE*. Proceedings of the IEEE, 79(9):1305-1320, 1991.

[13] S. Cheng, P.C. McGeer, M. Meyer, T. Truman, A. Sangiovanni-Vincentelli, P. Scaglia. *The V++ System Design Language*. Design Automation and Test in Europe, 1998.

[14] D.C. Luckham, J.J. Kenney, L.M. Augustin, J. Vera, D. Bryan, W. Mann. *Specification and Analysis of System Architecture Using Rapide*. IEEE Transactions on Software Engineering, Special Issue on Software Architecture, 21(4):336-355, April 1995.

[15] R. Piloty, M. Barbacci, D. Borrione, D. Dietmeyer, F. Hill, P. Skelly. *CONLAN - A Formal Construction Method for Hardware Desceiption Languages: Basic Principles*. National Computer Conference, Vol. 49, Anaheim, 1980.

[16] I. Joyner. C++?? : A Critique of C++ (3rd Edition). ftp://ftp.brown.edu/pub/c++/C++-Critique-3ed.PS.

[17] C. Liem, P. Paulin. *Compiler Techniques and Tools for Embedded Processor Architectures*. W. Wolf, J. Staunstrup, editors, Hardware/Software Co-Design Principles and Practice, Kluwer Academic Publishers, 1997.