Specification of Non-Functional Intellectual Property Components

Jianwen Zhu, Wai Sum Mong
Electrical and Computer Engineering
University of Toronto, Ontario M5S 3G4, Canada
{jzhu,mong}@eecg.toronto.edu

Technical Report TR-02-08-02
August 2002

Electrical and Computer Engineering
10 King's College Road
Electrical and Computer Engineering
University of Toronto, Ontario M5S 3G4, Canada

{jzhu,mong}@eecg.toronto.edu

Abstract

In its most general sense, intellectual property components (IPs) refer to any design artifacts that are reusable. While the specification of the functional IPs, such as behavioral and RTL specifications have been widely investigated, the specifications of others, such as timing, constraints, layouts and architectures are largely ad hoc. This leads to different standard or proprietary file/database formats with interoperatability problems, which eventually hinder the distribution and integration of IPs. In this paper, we address the difficult problem of integrating semantically diverse non-functional IPs by the use of a new, extensible language called Babel. Despite its simple 1-page grammar, Babel is frontend for a powerful IP-based design infrastructure. We demonstrate the effectiveness of our approach by two case studies, one for the creation of parameterized memory IPs and one for the creation of processor IPs.

Contents

1. Introduction

It is generally felt that the complexity involved in systems-on-chip design can only be tamed by intellectual-property (IP) based design. An IP can be intuitively interpreted as any piece of design artifact that is reusable in space (by other groups or companies) or time (in subsequent projects). An IP *infrastructure* that helps define and distribute these reusable artifacts is essential to a successful IP integration methodology. Without such an infrastructure, IP users tend to spend more time understanding the third-party IPs than creating their own. Without such infrastructure created as a *standard*, IP users tend to spend more time solving the interoperability problem than the design problem.

Existing standard bodies seem to focus more on the "standard", rather than the "infrastructure" aspect of the much needed infrastructure standard. Effort has been heavily invested in establishing an agreement on what the expected IP-deliverables are and what the corresponding file formats are. Not surprisingly, these standards reuse many *de facto* standard formats, often created by the dominant EDA vendors. While these formats are certainly adequate to capture the design information, the key problem is that they are bloated with information not relevant to the IP users. This leads to serious security problems, as the IP providers are not necessarily willing to reveal the design information.

With the observation of this *over-specification* problem, we created an experimental IP-infrastructure, called *ipsuite*, which focuses on the *interface* between the IP providers and IP users by abstracting away the design information irrelevant to the IP users, even though it may be essential to re-produce the IP. This interface is defined by a set of APIs, which we will elaborate in Section 3. While creating an IP, or using an IP, is as easy as writing C code that makes local or remote API calls in our infrastructure, this approach is often cumbersome for the IP vendors and therefore a language frontend is needed. Much effort has been devoted to the development of functional specification languages based on models of computations. However, such languages cannot be readily applied to the other important aspects of IPs, which are data-centric rather than computation-centric. Defining a universal language for the specification of non-functional IP is challenging in that the IP infrastructure needs to integrate information with completely different yet interacting semantics. To make things worse, the IP infrastructure, and therefore its frontend language, has to be extensible in order to accommodate the rather rapid advancement of technology.

In this paper, we demonstrate our experimental language, called Babel, to address this difficult challenge. The important goals we manage to achieve in this language are *simplicity*, since the definition of its grammar can be fit in only one page; *expressiveness*, since the type system of the language can be used to define data model of arbitary complexity; and *extensibility*, since new IP models can be defined without changing the language syntax.

In the sequel, we briefly review the related work in Section 2. While it is not the focus of this work to define our IP infrastructure, we give enough detail in Section 3 given its relevance. We describe in detail the Babel language in Section 4. In Section 5, we present two case studies, one for the creation of parameterized hard IPs, and one for the creation of processor IPs, under our IP-centric design environment.

2. Related Work

Since languages targeting system-on-chip design are badly needed, many proposals from academia, industry as well as standard bodies have emerged. The system level design language (SLDL) Rosetta [7], can be considered a natural step after IEEE's effort for standardizing VHDL/Verilog. Rosetta tries to establish a language substrate where different semantical domains, each of which represents a model of computation, can interact with each other. Another trend in SLDL is to leverage the legacy of software programming languages C/C++ by extending it to handle hardware as well. For example, CynApps announced its Cynlib [5], a C++

class library which provides features so that C++ can be used to model hardware. The Open SystemC Initiative, announced a similar library called SystemC [6] [9]. Another implementation with arguably superior simulation performance is the OCAPI library [8] developed by IMEC. The SpecC system level design language [12] supports protocol level component reuse, although it lacks the polymorphic type system desired. With its powerful type systems, the OpenJ language [13] has provided a language framework for experiments of system level design languages, although it did not explicitly define an RTL abstraction.

The Babel language is a complementary effort to SLDL research since it specifically addresses the specification of non-functional aspects of the reusable design components. In that sense, Babel shares much of its design goal and architecture with XML, an extensible language for internet document, which has become popular recently [11]. Other than being more concise, the advantage of Babel over XML in the context of IP specification is that Babel contains a very expressive type system, which allows the precise definition of data models. In addition, for IP integration tools, a "type sound" Babel specification is easier to process than untyped file formats.

3. IP Infrastructure

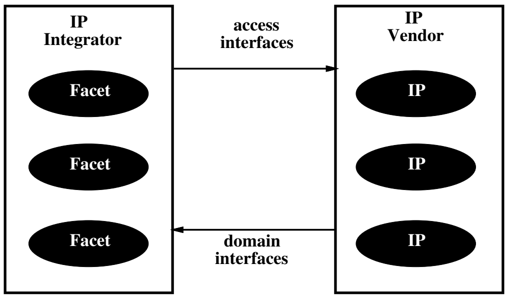

Figure 1. *ipsuite* **infrastructure.**

Many IP companies ran out of steam in a few years despite their promising starts. One of the fundamental reasons for a rather obviously viable business model to fail is the lack of standardized infrastructure support from the EDA industry. Today's design frameworks offered by EDA companies are monolithic: they consists of a set of proprietary tools manipulating a proprietary design database, typically mapped to the UNIX file system with tens of proprietary file formats. While the standardization efforts are underway in the semiconductor industry [10] to define IP deliverable formats, the cost of using an IP sometimes exceeds the cost of creating them, and hence IPs are rather referred to as "Intense Pain" by many [3]. This is in contrast to the software industry, where an application binary interface (ABI) is defined for every platform to make sure software libraries from different vendors can be linked to form an application. Middleware standards such as CORBA, DCOM and Java went one step further by allowing software components to collaborate across different platforms.

To address the above issues, we have created a new component-based EDA software infrastructure, called *ipsuite*. A wide departure from the traditional tool-centric EDA environment, *ipsuite* contains three types of primary elements which collaborates to design system-on-chip: the *facets*, the *domains* and the *IPs*, as shown in Figure 1. *IPs* in *ipsuit* are in the form of dynamically loadable software components provided at the IP *vendor* site. When loaded and invoked via an *access* interface, an IP create in memory a *facet* at the IP *integrator* site. Each facet represents a particular view of a design component such as behavioral facet and layout facet. IPs create facets by invoking standard APIs, called *domain* interfaces, each of which is associated with one type of facet. Table 1 summaries the characteristics and potential advantages over traditional design framework under the context of IP-centric design paradigm.

In our experimental infrastructure, the domain interfaces are defined in the form of COM interfaces, a language-independent binary standard defined by Microsoft [4]. The COM components can be dynamically loaded or even remotely loaded anywhere over the internet given enough middleware support.

We have defined and implemented the necessary domains to create different facets:

- behavioral facet, which captures the func-

Flexibility	Treating IPs software components allow bi-directional communcation between design tools and IPs.
Abstraction	IPs do not have to carry irrelevant implementation detail, which can be left to the integration framework.
Security	Firewall-like proctocols can be supported by the interfaces.
Deployability	IPs can be deployed by vendor using standard middleware framework.
Interoperability	Abstract and hence simple interfaces are easy to be standardized.

Table 1. *ipsuite* **features.**

tional view of a design component at the algorithm, as well as register transfer and logic level;

- planning facet, which captures the physical planning of a design component;
- mask facet, which captures the mask layout of design component;
- architecture facets, which capture the architectural information of the design component; one typical use is to help retargetting a compiler for a processor core.

While the development of *ipsuite* is still ongoing, a recent snapshot shows 5K lines of interface definitions, as well as 100K lines of C code implemenation, not including the CAD software packages we chose to reuse such as the Berkley magic layout database. In addition, retargetable compilers and high-level synthesis tools are being developed on top of this infrastructure.

4. Babel Langauge

4.1. Language Architecture

Instead of delivering functional IPs in source code form or non-functional IPs in proprietary formats, which reveal all the design information, the IP vendors now only need to ship small software components. While each of these software components can be created by compiling C code to make dynamic function calls to the domain interfaces, for many cases it is rather cumbersome to manually write the C code. A more user-friendly IP authoring methodology is to capture IP by a formal language, leaving the job of C code generation to a specialized IP compiler. We have developed such compilers for C, Java, Verilog as well as a new object-oriented SLDL called Wenyan into behavioral facets, yet there is no language for other non-functional IPs, hence the creation of Babel.

The unique and often conflicting requirements of non-functional IPs shape the design of Babel.

- Non-functional IPs model complex data, rather than complex function.
- There is no way to unify the semantics of non-functional IPs.
- The interactions of non-functional IPs need to be specified on a common language substrate for simplicity.

Figure 2 shows the architecture of the Babel system, including its compiler. In contrast to the traditional language design where a syntax is devised for a specific semantics, we designed the syntax for a *meta-semantics*, where a *type system* is devised so that it is expressive enough to define the data model for each facet type. Typically, a data model is provided in a header file as a set of type definitions. Such header files can be included in IP specifications, which consist of data using the *expression system* of the language. A Babel compiler includes a parser which translates an IP specification into an intermediate representation, after which a type inference engine is invoked to ensure the soundness of the specification. Any specification resulting in a type error, in other words, violating the defined data model, will be dismissed. The type-checked intermediate representation will be fed into a dynamically loadable software component, called *domain plugins*, to be further compiled into the IP form acceptable to the *ipsuite* infrastructure. Babel language is said to be extensible in the sense that whenever a new domain is needed, one only has to add a header file containing the data model, as well as the corresponding domain plugin, while keeping the language syntax and compiler proper intact.

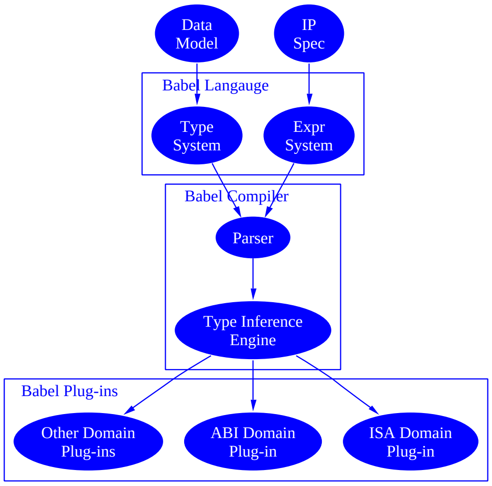

Figure 2. Babel language architecture.

As specified in the BNF grammar shown in Figure 3, a Babel specification consists of a set of enumerate type declarations (*enumDecl*) and class type declarations (*classDecl*) to specify new types, and a set of facet declarations (*facetDecl*) to specify the IPs. For convenience, aliases to types can be declared as well (*aliasDecl*). Class declarations are typically used to specify the data model of a facet. Like any object-oriented languages, a class contains a set of fields and methods. In addition, classes in Babel can be polymorphic, in the sense it can be parameterized over other types or constants (*genericDecl*).

```
babel  ::= (enumDecl | classDecl | aliasDecl | facetDecl)*
enumDecl ::= "enum" ID "{" (ID "[" "=" exprBinary "]")* "}"
classDecl ::= "class" ID [genericDecl] classBody
genericDecl ::= ( "class" ID | type ID )*
classBody ::= "{" (classDecl | fieldDecl | methodDecl)* "}"
fieldDecl ::= type ID [ "=" expr ] ";"
methodDecl ::= type ID "(" [parameters] ")"
(";" | "{" "return expr ";" "}")
parameters ::= type ID ( "," type ID )*
aliasDecl  ::= "typedef" type ID ";"
facetDecl  ::= "facet" ID "::" ID "=" exprCons
```

Figure 3. Declarations.

4.2. Type System

In addition to the predefined primitive types (*typePrimary*) and the elaborated class types (*typeClass*), one can construct complex types not found in traditional languages, as specified by the type system grammar in Figure 4. A type in Babel can be considered as a set. One can thus define mathematical concepts such as function (*typeFunction*), union set (*typeUnion*), Cartesian set (*typeProduct*), sequence and power set (*typeUnary*).

```
type        ::= typeFunction
typeFunction::= typeUnion "->" typeUnion
typeUnion   ::= typeProduct "|" typeUnion
typeProduct ::= typeUnary "^" typePoduct
typeUnary   ::= ["[]" | "<>" ] (typePrimary | typeClass)
typePrimary ::= [unsigned ] int | [unsigned] long |
                [unsigned] char | float | double
typeClass   ::= ID [("." ID)*]
                   [ "[" typeParam ("," typeParam)* "]"]
typeParam   ::= type | exprBinary
```

Figure 4. Types.

Example 1 *A data model specification fragment.*

```
typedef int^int      Cell;      // a pair of integers

typedef []field     CellGroup;  // a sequence of fields

class Store {
  Store( int gran, int size ); // required properties
                               // optional properties
  int         depth;
  int         overlap;
  field       pointer;
  {}CellGroup cells;           // a set of groups
  }
```

4.3. Expression System

An IP is specified in the facet declaration (*facetDecl*) construct whose content is specified as a *constructor* expression (*exprCons*). A constructor expression creates a complex graph of data conforming to the data model specified by a class type. The content of a constructor is a hierarcical list of statements (*exprStmt*). A statement can either be a field assignment, where the field has to be declared in the corresponding class; or a method call, where the method has to be declared in the corresponding class. Typically, the method body of a class body is empty. It is up to the domain plug-ins to interpret the semantics of the method calls. One can also specify conditional statements or generate statements for more complex data. An expression can also be specified hierarchically with binary or unary operators in the same way as traditional programming languages (*exprBinary, exprUnary*); it means that the leaves can either be a constant, a set, a sequence, or a tuple. Note that each expression can be prefixed with an alias, which can later be referenced. In this way, an expression not only can specify a tree of data, but also a graph of data.

Example 2 *An IP specification fragment.*

```
stores = {
    sGPR = new Store( 32, 32 ) {
      depth = 128;
      overlap = 24;
      pointer = cwp;
      maps = {
          gpr = [
              g0, g1, g2, g3, g4, g5, g6, g7,
              o0, o1, o2, o3, o4, o5, o6, o7,
              l0, l1, l2, l3, l4, l5, l6, l7,
              i0, i1, i2, i3, i4, i5, i6, i7
              ]
          alias = [
              x, x, x, x, x, x, x, x,
              x, x, x, x, x, x, sp, x,
              x, x, x, x, x, x, x, x,
              x, x, x, x, x, x, fp, x
              ]
          }
      }
    ...
    }
```

4.4. Domain Plug-ins

Babel is very much like functional languages where types of expressions can be automatically determined by the type inference engine of its compiler. This ensures that IPs can be captured in a concise fashion. The type checker ensures that the specification is free of type errors, in other words, conforms to the required domain data model. Note that the type inference engine and type checker can eliminate most of the mundane modeling errors. Both of them can be shared by all domains. The domain plug-ins translate each parsed and checked Babel specification into the raw C code, which contains calls to the corresponding domain interfaces, and can be compiled into non-functional IPs accepted by the *ipsuite* infrastructure.

```
exprCons    ::= "new" typeClass [ expr ("," expr)* ]
                    [ "(" expr ( "," expr )* ")" ]
                    ( ";" | "{" (exprStmt)* "}" )
exprStmt    ::= ID "=" expr ";" |
                ID [expr ("," expr)* ";" |
                "if" "(" exprBinary ")" exprStmt
                    ["else" exprStmt]   |
                "generate" "(" ID "," expr "," expr ")"
                    exprStmt
expr        ::= "new" exprCons | exprBinary
exprBinary  ::= exprUnary ("+" | "-" | "*" | ...) exprBinary
exprUnary   ::= ("*" | "~" | "!") exprPrimary
exprPrimary ::= literal | ID | type |
                [typeClass "." ] ID
                [ "(" [type ("," type)*] ")" ] |
                exprList | exprSeq | exprSet | exprTuple
exprSeq     ::= "[" [ID "="] expr ("," expr)* "]"
exprSet     ::= "{" [ID "="] expr ("," [ID "="] expr)* "}"
exprTuple   ::= "<" [ID "="] expr ("," [ID "="] expr)* ">"
exprList    ::= "(" [ID "="] expr ("," [ID "="] expr)* ")"
```

Figure 5. Expressions.

5. Case Studies

In this section, we present two case studies. The first case study demonstrates how *ipsuite* infrastucture can be directly interfaced to create non-functional IPs. The second case study demonstrates how Babel can be used to create non-functional IPs in a more efficient way.

5.1. Parameterized Memory Cores

A large class of SOC components have well-defined, regular layout structures which render the synthesis methodology unsuitable. Such components include ROMs, PLAs, SRAMs, FIFOs, and datapaths. For these circuits, the designer often has an algorithm in mind to combine basic components, usually by *tiling*, into a layout structure. Such algorithms are designed to work with different parameters. Since components have their own specific algorithms, which are difficult to generalize, it is impossible to write a tool that can generate them all. A better and extensible approach is to implement each component in the form of *ipsuite* IP, which itself is a software component. The *ipsuite* defines two interfaces, `IMaskTech`, which abstracts the technology information such as design rules, and `IMask`, which defines the API for plain layout generation. A C++ wrapper of these two interfaces, called `SimpMask`, is created to further simplify the usage. In addition, `SimpMask` defines the tiling abstraction to simplify the job of layout composition.

We used `SimpMask` to create a small SRAM. We reused 58 custom designed leaf cells originally developed in Berkeley's low-power cell library [1], including the 6-T storage cell and the sense amplifier. We use a polymorphic class to represent a SRAM component, which contains parameters such as the number of words and the number of bits. The SRAM IP is hierarchically defined using 6 C++ files, each of which accesses *ipsuite* domain interfaces via `SimpMask`. With only 665 lines of C++ code, we are able to generate a parameterized SRAM IP. Figure 6 shows one instance of this IP. With this case study, we believe that the traditional silicon compilers fit very well into the *ipsuite* paradigm.

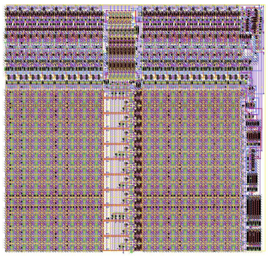

Figure 6. SRAM IP.

5.2. Processor Cores

Not all IPs can be conveniently captured by C/C++ programs. In this case study, we demonstrate how Babel can be used to create processor

Processor	BEH facet (#lines)		ISA facet (#lines)		ABI facet (#lines)	
	Babel	C	Babel	C	Babel	C
SPARC	196	2892	2300	7245	56	563
SimpleScalar	90	1598	1048	3244	45	633
Alpha	1144	9015	4511	11039	52	643

Table 2. Architectural IPs using Babel.

IPs. While the common perception of a processor core is its RTL, a truly reusable processor core should also contain the software development environment including compilers, assemblers, linkers, and instruction set simulators. In order for these tools to be automatically retargetted to the core, abstract architectural information has to be supplied, which we refer to as *architectural IPs*. We define several domains, *ISA, ABI* and *MICRO* to capture the instruction set, application binary interface, and instruction-level parallelism respectively. For each domain, a Babel data model is defined using the Babel type system. The architectural information of the processor core can then be captured using the Babel facet construct.

Table 2 shows the complexity of Babel specifications for SPARC processor, SimpleScalar processor and Alpha processor. For each processor, we show the size of its behavioral, ISA and ABI facets in terms of the number of lines of code. Both the size of Babel specification and the size of C code generated by corresponding Babel domain-plugins are shown. It is evident that the generated C codes are many times larger than its Babel counterpart.

Our integration framework contains a set of automatic retargeting tools. Our `rbinutils` tool is used to automatically port GNU's binary utilities `binutils`. The package contains a wide range of production-quality binary tools designed to manipulate object files for example, the GNU assembler `gas` and the GNU link editor `ld`. The package is also highly complex with a daunting size of a quarter million lines of C code. The package contains several components that are machine-dependent, including `BFD` library, which is a foundation library for object file manipulation, `opcodes` library, which is a library for instruction disassembling and assembly parsing, `gas`, which is an assembler, and `ld`, which is a link editor.

File	#line
BFD library	12361
opcodes	3185
gas	6424
ld	997
total	22967

Table 3. Machine dependent code generated.

Benchmark	Simulation runtime (s)	Native runtime (s)
181.mcf	338.47	1.22
197.parser	1480.69	4.46
183.equake	2143.50	6.20
188.ammp	12774.84	36.66

Table 4. Runtime of generated simulator.

Our tool can automatically generate the machine-dependent components from the information provided by the architectural IP, thereby porting the `binutils` package for the corresponding processor core. Table 3 shows the machine-dependent code automatically generated for different components of the package for the SPARC processor. We verified the methodology by comparing the executable produced by the generated binary tools against the manually developed ones distributed by GNU.

We have also developed `rscalar`, a tool that can automatically port SimpleScalar [2], a popular instruction set simulator developed at University of Wisconsin. SimpleScalar was originally developed only for one instruction set. Our tool leverage its rich set of micro-architecture simulation components, such as those for memory, cache, scheduler and branch predictor, while automatically generates the machine dependent part according to the architectural IP provided. Table 4 shows the simulation runtime of the SPEC2K benchmark using our generated simulator for the SPARC processor. The simulation is performed on a 750MHz SunBlade 1000 workstation.

6. Conclusion

We believe an IP-infrastructure needs to be investigated before a comprehensive IP standard is finalized. Such IP-infrastructure needs language frontends which not only specify the functionality, as many of the previous efforts, but also other aspects of design that need to be reused. The proposed Babel language works very well with our experimental component-based IP infrastructure.

7 References

[1] T. Burd. Very low power cell library. Technical report, University of California, Berkeley, 1995.

[2] D. Burger and T. M. Austin. The SimpleScalar tool set, version 2.0. Technical Report #1342, University of Wisconsin-Madison, Computer Science, June 1997.

[3] J. Chilton. IP reuse quality: Intelletual property or intense pain. In *Proceedings of the International Symposium on Quality Electronic Design*, San Jose, USA, March 2002.

[4] *Microsoft Component Object Model*. `http://www.microsoft.com/com`.

[5] *CynLib Web Site*. `http://www.cynapps.com/CynApps/products/cynlib/opensource.html`.

[6] S. Liao, S. Tjiang, and R. Gupta. An efficient implementation of reactivity for modeling hardware in the Scenic design environment. In *Proceeding of the 34th Design Automation Conference*, 1997.

[7] *Rosetta Web Site*. `http://www.sldl.org/`.

[8] P. Schaumont, R. Cmar, S. Vernalde, M. Engels, and I. Bolsens. Hardware reuse at the behavioral level. In *Proceeding of the 36th Design Automation Conference*, New Orleans, June 1999.

[9] *SystemC Web Site*. `http://www.systemc.org`.

[10] *Virtual Socket Interface Alliance*. `http://www.vsi.org/`.

[11] *Extensible Markup Language (XML) 1.0*. `http://www.w3.org/TR/1998/REC-xml-19980210`, February 1998.

[12] J. Zhu, R. Doemer, and D. G. Gajski. Syntax and semantics of SpecC language. In *Proceedings on the Seventh Workshop on Synthesis and System Integration of Mixed Technologies*, Japan, December 1997.

[13] J. Zhu and D. G. Gajski. OpenJ: A system level design language. In *Proceedings of the Design Automation and Test Conference in Europe*, Munich, Germany, March 1999.