Reuse and Protection of Intellectual Property in the SpecC System

Rainer Dömer, Daniel D. Gajski
{doemer,gajski}@ics.uci.edu
Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425, USA

Abstract—
In system-level design, the key to cope with the complexities involved with System-on-Chip (SOC) designs, is the reuse of Intellectual Property (IP). With the increasing demand for IP, the mechanism to protect an IP component from being copied, modified, or reverse-engineered, becomes very important. This paper describes how reuse and protection of IP is supported by the SpecC language and the SpecC design environment.

I. Introduction

The semiconductor roadmap estimates the design complexity for digital systems to continue to increase according to Moore's law. In the next years, systems with 10ths of millions of transistors on one chip will be standard technology. System-on-Chip (SOC) designs will integrate processor cores, memories and special-purpose custom logic into a complete system fitting on a single die. However, the increased complexity also requires more effort, more efficient tools and new methodologies for building such designs. Increasing the design time is not an option due to the market pressures.

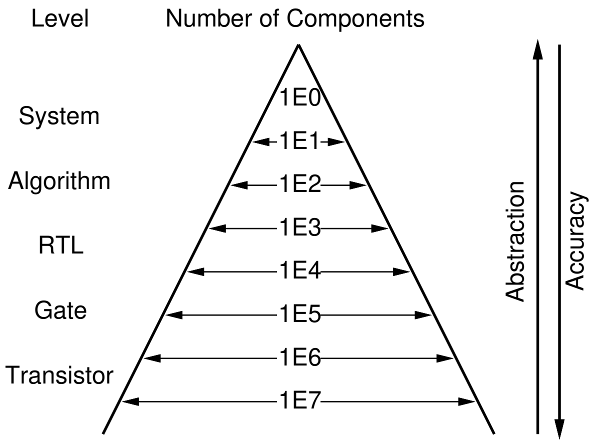

Fig. 1. Abstraction vs. Complexity

In computer science, a well-known solution for dealing with complex systems is to use a hierarchical approach and to move to higher levels of abstraction. This effectively reduces the number of components to be handled in each task.

Figure 1 illustrates this. A system, which at the transistor level is composed of 10ths of millions of transistors, typically reduces to only thousands of components at the register-transfer level (RTL). Furthermore, RTL components are grouped together at the algorithm (or behavioral) level. Finally, at the system level, the one system is composed of only few components, including processing elements (PEs), memories and busses.

A top-down design methodology starts with a specification at the highest level of abstraction, the so-called system level, and step-wise moves down to lower levels refining the model. With each step, the design becomes a more and more accurate model of the final implementation. On the other hand, a bottom-up methodology starts by using components from the lowest level, composing them together. These composed components then can be used in the next step to build even more complex components.

Both methodologies can be combined in order to achieve the best productivity. Usually, the top-down approach is applied first until the system is decomposed into components which can be selected from a component library. The component library is built using the bottom-up strategy.

Since the *time-to-market* is crucial for a product, it must be emphasized that only the top-down design time applies, because the component library can be built beforehand (possibly by somebody else). Thus, the key to a short design time enabling *"product on demand"* is the use of predesigned, complex components which can be easily integrated in order to build the product. Such components are called *Intellectual Property* (IP) and the system design methodology, which is based on the integration of IP components, is called *IP-centric* [2].

Typical IP components include memories, processors (general purpose as well as application specific ones like DSPs), and special purpose circuits for standard applications like encoding/decoding algorithms and communication protocols (e. g. a PCI-bus interface). It should be noted that IP includes both software and hardware components.

Since the process of developing the system is decoupled from the development of its components, these tasks can be performed by different companies. While system houses focus on the problem of system specification, integration and implementation, IP vendors develop and provide the required IP components. With this approach, the system house benefits from a large library of optimized, well-tested and well-documented components, while the IP providers can take advantage of their expertise in specialized areas.

While the IP-centric methodology promises great benefits, there are also problems to be solved. This paper addresses the problems of IP modeling for reuse and IP protection. Both problems are solved with the help of the SpecC language [7] and are implemented in the SpecC design environment [8].

The rest of the paper is organized as follows: After a brief discussion of related work, Section II describes how IP components must be modeled in order to be reusable and how this is done with the SpecC language. Then, Section III introduces the protection mechanism which allows the use of IP without revealing its internal implementation. Experiments and results are shown in Section IV. Finally, Section V concludes the paper with a brief summary and description of future work.

A. Related work

Modeling and protection of IPs is different for hard IPs and soft IPs [5]. Hard IP components are developed by use of a standard design process and the final implementation is not given to the system integrator. Instead, simulation and synthesis models of the IP are used by the system integrator together with documentation. With this method, the IP is protected because its implementation stays with the IP provider. For soft IPs, whose final implementation will be synthesized, the complete model is needed by the system integrator. For protection of the implementation and algorithm details, the IP can be provided in precompiled format without source code.

In addition, Watermarking [4] can be used to protect an IP by insertion of a hidden watermark which ensures that the IP can always be identified.

A different approach is to leverage the recent advancements in Internet technology. For example, when using Java as simulation language [1], IP components can stay at the providers site and simulation can be performed via the Internet. Although such an approach is interesting and very safe, it suffers from the dependency on the network in terms of availability and transfer speed.

II. IP-centric Modeling

For the system integrator, IP components need to be modeled in a way so that reuse, selection and integration becomes easy. In order to allow such *"plug-and-play"* with IPs, the IP model must be simple, versatile, and well-defined in terms of its functionality and its interfaces.

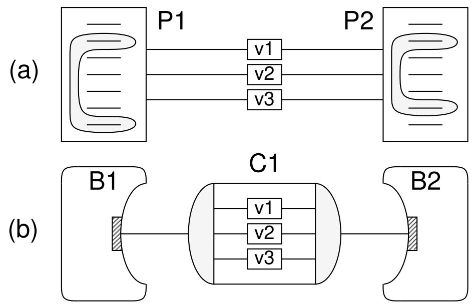

Fig. 2. Separation and encapsulation of communication and computation

The essential requirements are *separation* and *encapsulation* of communication and computation, as illustrated in Figure 2. A typical model of two communicating processes described in VHDL or Verilog, for example, is shown in Figure 2(a). The code of the processes contains both communication and computation freely intermixed. In such a model, there is no way to automatically change the communication protocol when the connecting bus is replaced, because the code for communication cannot be identified.

In the SpecC model [7], as shown in Figure 2(b), this problem does not exist because the computation is encapsulated in *behaviors* (`B1`, `B2`), and the communication is contained in *channels* (`C1`). Hence, the communication part can be clearly identified and the channel can be easily replaced by a different channel which provides the same interfaces.

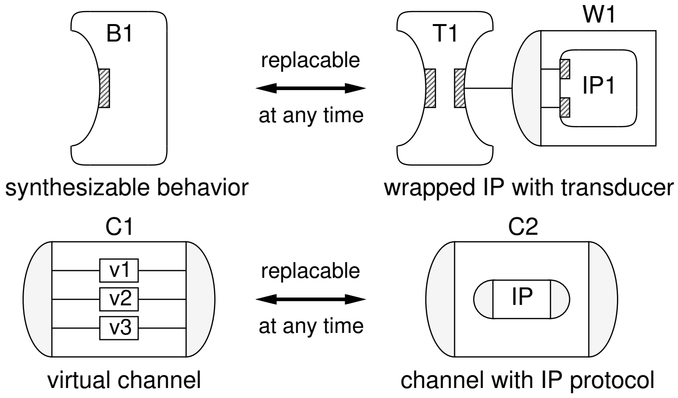

Fig. 3. Plug-and-Play with IPs

Naturally, the SpecC model also allows the hierarchical composition of behaviors and channels in terms of both structure and behavior. Figure 3 summarizes the "plug-and-play" feature supported by this model. At any time in the design cycle, behaviors and channels can be replaced with IP components which, if necessary, are wrapped in channels providing interface adaption and protocol conversion.

In order to demonstrate how behaviors, channels and interfaces are specified with the SpecC language, a more detailed example is described next. Figure 4 shows a graphical representation of the following SpecC source code.

```
1 interface I1
2 {
3       bit[63:0] Read(void);
4       void Write(bit[63:0]);
5 };
6
7 channel C1 implements I1;
8
9 behavior B1(in int, I1, out int);
10
11 behavior B(in int p1, out int p2)
12 {
13 int    l1;
14 C1     c1;
15 B1     b1(p1, c1, l1),
16        b2(l1, c1, p2);
17
18 void main(void)
19   { par { b1.main();
20           b2.main();
21         }
22   }
23 };
```

The example specifies a behavior `B` which is composed of two concurrent executing subbehaviors `b1` and `b2`. These are interconnected by a local variable `l1` and a communication channel `c1`. The communication protocol implemented in the channel `C1` is specified in interface `I1`.

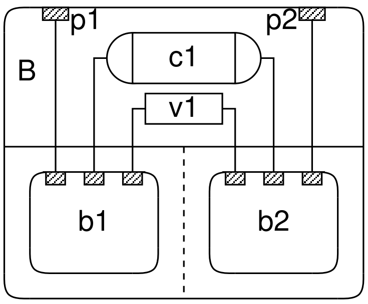

Fig. 4. Example model in SpecC

Note that channel `C1` in line 7 and behavior `B1` in line 9 are declared with their interfaces but have no implementation (no body is defined). The actual implementation of the communication protocol in `C1` and the functionality of `B1` is hidden. For simulation purposes, it can be supplied by a library to be linked with the simulation executable.

Also, please note how "plug-and-play" works with both behaviors and channels. Given a behavior `B2` (or channel `C2`) with a different implementation but with compatible ports (interfaces), it is just a matter of replacing `B1` with `B2` in line 15 (`C1` with `C2` in line 14) in order to switch to a new component.

III. IP Protection

The approach taken for IP protection in the SpecC system is based on the idea of providing the secret implementation in form of a precompiled library. The public interfaces of the IP component are specified by use of behavior, channel, and interface declarations.

This is basically the same approach as used for software. Reuse of software components means usually a set of function and variable declarations, whose implementation is supplied by a linker library. All the necessary information to use such a software package is contained in the API declaration and the accompanying documentation. The actual implementation is hidden in object code and therefore protected.

However, for IP components modeled in SpecC, special care has to be taken to make sure a component cannot be reverse-engineered from the data made available. The following sections describe how this is achieved.

A. Public IP interface

In the SpecC system, components are of two types, behaviors containing computation, and channels encapsulating communication, as described earlier.

In analogy to functions in C, behaviors and channels consist of a *declaration* and a *definition*. The declaration specifies ports and interfaces, whereas the definition contains the actual implementation. For an IP component, the declaration is supplied in form of source code and the definition is provided as a precompiled library.

For behaviors, a typical declaration specifies the name of the behavior and the number and type of its ports. For example,

```
behavior IP1(in int P1, out bit[7:0] P2);
```

specifies a behavior `IP1` with one input and one output port. In addition, annotations can be attached to the behavior if necessary. For example:

```
note IP1.Version = 1.2;
note IP1.Area = 42000;
```

In order to declare a channel, its interfaces have to be defined first. For example, two interfaces describing send and receive methods for bytes and words of data can be defined as follows.

```
typedef bit[ 7:0] byte;
typedef bit[63:0] word;

interface I1
{
```

```
  void SendByte(byte B);
  byte ReceiveByte(void);
}

interface I2
{
  void SendWord(word W);
  word ReceiveWord(void);
}
```

With these definitions, a channel `IP2` implementing both interfaces can be declared as

```
channel IP2 implements I1, I2;
```

Of course, the channel `IP2` and its interfaces can be annotated in the same way as the behavior `IP1`.

It should be mentioned, that in the SpecC language actually both, behaviors and channels, can have ports and interfaces as well. The separation above is made simply for ease of understanding. Please note that this makes no difference to the applicability of the issues discussed in this paper.

B. Secret IP library

As mentioned above, the implementation of an IP behavior or IP channel is supplied as a precompiled library. This ensures that the secret implementation is hidden from the IP user. Furthermore, for hard IPs, this library only contains a simulation model and therefore it is not possible to synthesize the IP from the library.

In order to build such a library, the IP provider implements a class body for the behavior or channel and compiles it into a library. For example, for the behavior `IP1` in Section II, a shared library `libIP1.so` can be created.

However, the generation of such a library is not trivial because of the way behaviors and channels are implemented in the SpecC system. From SpecC source code, the SpecC compiler generates C++ code which will then be compiled by a standard C++ compiler in order to produce an executable file for simulation. Behaviors and channels are implemented as C++ classes and their instances are naturally represented by objects. Among other reasons, which are beyond the scope of this paper, this implementation was chosen because it keeps the generated code very similar to the original SpecC code and thus simplifies source-level debugging.

For example, a fragment of generated C++ code is shown next to the original SpecC code:

```
behavior B(                    class B: public Bhvr
 in  int p1,                   {
 out int p2)                    int &p1;
{                               int &p2;
 int l1;                        int l1;
 C   c1;                        C   c1;
 B1  b1(p1, l1, c1);            B1  b1;
 B2  b2(p2, l1, c1);            B2  b2;
 void main(void)                void main(void);
 {                              B(int &p1, int &p2);
  par{b1.main();                ~B(void);
      b2.main();}               };
 }
};
```

In C++, in order to instantiate a class, the size of the class must be known so that sufficient memory can be allocated for the new object before the constructor of the class is called to initialize the memory. While the constructor is provided in the class itself, the memory must be allocated by the instantiator. C++ semantics enforce that a class is defined (not just declared) before it can be instantiated, thus the size of the required memory is known when an object is created.

In the case of an IP component, which is supplied in a library, the size of the class still must be known by the user code. Therefore, in the C++ user code a class declaration as in SpecC is not sufficient. Instead a class definition is required. This is a problem for the IP user because he does not know the internals of the IP class and thus cannot create a proper class definition.

The problem can be solved if the size of the class is known. With this information, the IP user can create a pseudo class which only contains known contents and leaves enough space for the secret internals. In particular, this pseudo IP class consists of the known ports, the public interfaces and sufficient space reserved for the secret parts of the IP.

For example, a pseudo class for the behavior `B` listed above can be defined as

```
class B: public Bhvr
{
int &p1;
int &p2;
char Reserved[X];

void main(void);
B(int &p1, int &p2);
~B(void);
};
```

where the array `Reserved[X]` replaces the secret IP components `l1`, `c1`, `b1`, and `b2`. The array size `X` must be equal to (or greater than) the size of all the replaced components.

Please note that such a class replacement is highly compiler dependent since the C++ language [6] leaves some freedom for the implementation of classes. Therefore, when this approach is implemented, it must be integrated with the compiler being used.

With this solution, the IP component can be used as any other component, given the reserved size `X` is provided with the component declaration and the IP library. The value of `X` can be computed by the IP provider from the IP implementation.

In particular, the size of an IP class C is computed as $\text{sizeof}(C) = X_{public} + X_{secret}$, where

$$X_{public} = \sum_{p \in Ports(C)} \text{sizeof}(p) + \sum_{i \in Itrfcs(C)} \text{sizeof}(i)$$

$$X_{secret} = \sum_{l \in Locals(C)} \text{sizeof}(l) + \sum_{c \in Chnls(C)} \text{sizeof}(c) + \sum_{b \in Bhvrs(C)} \text{sizeof}(b) + \Delta$$

Here, Δ is some implementation dependent overhead needed for data alignment, etc. These values can be easily computed by the SpecC compiler since the sizeof() operator can always be evaluated at compile time.

IV. Experiment

The approach for protection of IP, as described in Section III, has been implemented and integrated with the SpecC compiler.

A. Implementation

The program flow of the SpecC compiler `scc` is illustrated in Figure 5. The default flow starts on the top with the SpecC source code of a design which is first processed by the Preprocessor and then fed into the SpecC Parser which builds a complex data structure, called SpecC Internal Representation (SIR).

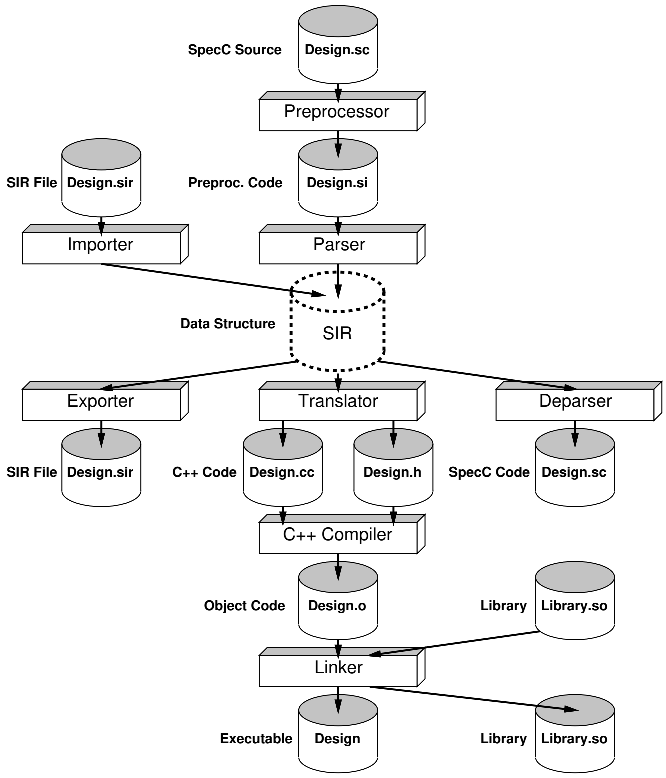

Fig. 5. Program flow of the SpecC compiler.

By default, the compiler generates C++ code from the SIR data structure which then is compiled and linked with the standard SpecC libraries to create an executable file for simulation. The SpecC compiler is also able to import and export binary SIR files and can even re-generate SpecC source code from the internal representation.

In order to support IP, the SpecC compiler has been extended with an *IP mode* (enabled by option `-ip`) which changes the behavior of the Exporter, the Deparser, the Translator and the underlying C++ Compiler and Linker (see Figure 5).

In IP mode, the compiler recognizes special annotations (`scc_Public`) which the user attaches to behaviors and channels to mark them as IPs with public ports and interfaces. All objects not marked public will be treated as secret implementation by the compiler and will be hidden.

For example, the Exporter and the Deparser will only generate code for the public objects, all other objects will be omitted. From the implementation of an IP, the IP provider can use this to automatically generate the files describing the public interfaces of his IP. Furthermore, when these public files are generated, the behavior and channel IP declarations will be automatically annotated with the reserved size (`scc_ReservedSize`), as discussed in Section B. This annotation will later be used as the value `X` in the IP pseudo classes generated by the compiler when the IP component is instantiated.

The compilation flow is also affected by the IP mode. When generating C++ code, the SpecC compiler ensures that only objects marked public will have external linkage. In other words, all non-public objects will have internal linkage and are therefore not visible outside the file scope. Also, the C++ compiler and the linker are instructed to create, instead of an executable file, a shared library for which all internal IP symbols are stripped off.

In summary, using the IP mode, the IP provider can automatically create the public IP interface and the IP library while being sure that no information about the implementation of the IP will be available to the IP user. On the other hand, the IP user can simply include the annotated interface declarations in his design and use the IP components just as his own behaviors and channels by linking his executable file against the provided IP libraries.

B. Design examples

Several example designs have been successfully tested with the implemented IP support. First, as a simple example, a generic adder has been specified at the gate level and has been modeled as an IP component in three different bitwidths. Then, the proposed IP protection scheme has been applied to four system-level designs of industrial size. These examples consist of two controller components, namely an elevator controller and a traffic light controller, and two data compression IPs, namely a JPEG encoder and a GSM vocoder. The vocoder alone consists of about 13000 lines of SpecC source code [3].

The following table shows the characteristics of the IP models. In particular, the table lists the number of internal components and the reserved size for each IP.

IP example	Components	Reserved size
Adder, 8 bit	65	2428
Adder, 16 bit	131	5020
Adder, 32 bit	261	10052
Elevator controller	91	4248
Traffic light ctrlr.	24	892
JPEG encoder	4	2728
GSM vocoder	84	12020

It should be mentioned that, in contrast to the internal structure, the reserved size is visible for the IP user. In order not to reveal the complexity of the IP implementation through this number, an IP provider is free to choose any number greater than the minimum computed by the compiler. For example, the reserved size 12000 works well for all the adders.

Using the IP-enabled SpecC compiler, a public interface and a shared library have been created automatically for all IPs. For example, the public interface generated for the GSM vocoder is shown next:

```
//////////////////////////////////////
// SpecC code generated by scc V2.0.4
// Design: GSM_Vocoder_public.sc
//////////////////////////////////////
behavior Coder(
        in bit[12:0] Sample,
        out unsigned bit[243:0] Frame,
        in bool DTX_Mode,
        out unsigned bit[5:0] DTXctrl,
        in event NewSample,
        out event FrameReady );
note Coder.Version = "GSM_06.60";
note Coder.scc_ReservedSize = 12020u;
//////////////////////////////////////
```

V. Conclusion and Future Work

System-on-Chip design must be based on the reuse of IP. In order to support IP, the models used in system design must naturally integrate IP components and allow "plug-and-play". This requires the clear separation of computation and communication which is directly supported by the SpecC model with behaviors and channels.

IP reuse and IP protection, as described in this paper, have been implemented in the SpecC design environment which has been made freely availabe on the Internet [8]. In particular for this paper, the SpecC compiler `scc` has been extended in order to support the recognition and use of IP components. Furthermore, the compiler supports the automatic generation of public IP interface files and secret IP libraries for any design specified with the SpecC language.

An IP library, generated automatically by this approach, will not reveal any information about the implementation of the IP. This ensures that these IP libraries are safe and fully protected against reverse-engineering but can be used just as any other component in the system.

The approach has been successfully tested with several examples, including industrial-sized systems such as a JPEG encoder and a GSM vocoder.

In conclusion, the SpecC design environment has been extended to support an IP-centric methodology with easy IP reuse and automatic IP protection.

Future work will focus on IP-based architecture exploration and communication synthesis.

References

[1] M. Dalpasso, A. Bogliolo, L. Benini. "Specification and validation of distributed IP-based designs with JavaCAD". In *Conference Proceedings of Design, Automation and Test in Europe*, Munich, Germany, Mar. 1999.

[2] D. Gajski, R. Dömer, J. Zhu. "IP-centric Methodology and Design with the SpecC Language". In *System Level Synthesis*. Edited by A. Jerraya, J. Mermet. Kluwer Academic Publishers, 1999.

[3] A. Gerstlauer, S. Zhao, D. Gajski, A. Horak. *Design of a GSM Vocoder using SpecC Methodology.* Technical Report ICS-TR-99-11, University of California, Irvine, Feb. 1999.

[4] A. Kahng, J. Lach, W. Mangione-Smith, S. Mantik, I. Markov, M. Potkonjak, P. Tucker, H. Wang, G. Wolfe. "Watermarking Techniques for Intellectual Property Protection". In *Proceedings of the Design Automation Conference*, San Francisco, 1998.

[5] M. Keating, P. Bricaud. *Reuse Methodology Manual for System-on-a-Chip Designs.* Kluwer Academic Publishers, 1998.

[6] B. Stroustrup. *The C++ Programming Language*, 3rd edition. Addison-Wesley, 1997.

[7] J. Zhu, R. Dömer, D. Gajski. "Syntax and Semantics of the SpecC Language". In *Proceedings of the Workshop on Synthesis and System Integration of Mixed Technologies*, Osaka, Japan, Dec. 1997.

[8] `http://www.ics.uci.edu/~specc/`