The RAMP Architecture & Description Language

Greg Gibeling, Andrew Schultz & Krste Asanović
RAMP Gateware Group, UC Berkeley & MIT CSAIL
{gdgib & alschult}@eecs.berkeley.edu, krste@csail.mit.edu

February 16, 2006

1 Introduction

The RAMP (Research Accelerator for Multiprocessors) project is developing infrastructure to support high-speed emulation of large scale, massively parallel multiprocessor systems using FPGA platforms. In this paper, we describe the goals and implementation of the RAMP Design Framework (RDF). The framgework must support both cycle-accurate emulation of detailed parameterized machine models and rapid functional-only emulations. The framework should also hide changes in the underlying RAMP implementation from the designer as much as possible, to allow groups with different hardware and software configurations to share designs, reuse components and validate experimental results. In addition, the framework should not dictate the implementation language chosen by developers.

Our approach was to develop a decoupled machine model and design discipline, together with an accompanying RAMP Description Language (RDL) and compiler to automate the difficult task of providing cycle-accurate emulation of distributed communicating components.

The RAMP Design Framework is structured around loosely coupled units, implemented in a variety of technologies, communicating with latency insensitive protocols over well-defined channels. This paper documents the specifics of this framework including the interfaces that connect units, and the functional semantics of the communication channels. We cover the RAMP architecture, description language and compiler in detail, but avoid wherever possible those details which can and should vary on an implementation basis.

In section 3 we describe the interfaces and semantics of the highest abstraction in RAMP, that of the system being emulated. We proceed, in section 4, to describe the implementation of this high-level abstraction in very general terms, avoiding platform and language specific constructs where possible. The remaining section 5 covers the RAMP description language and the compiler respectively and are accompanied by a glossary of RAMP terms found in appendix A and the RDL language reference in appendix B.

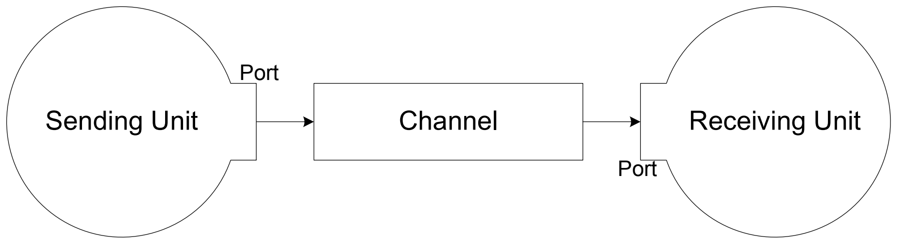
2 RDF Overview

The purpose of the RAMP Design Framework is to enables high-performace simulation and emulation of large scale, massively parallel systems on a wide variety of implementation platforms. For the RAMP project, the designs of interest will typically be collections of CPUs connected to form cache-coherent multiprocessors. In RDF the design of interest, *e.g.*the one being emulated, is refered to as the the *target* whereas the machine performing the emulation, *e.g.*a BEE2, is the *host*.

A RAMP target design is structured as a series of loosely coupled *units* which communicate using latency-insensitive protocols implemented by sending *messages* over well-defined *channels*. Figure 1 gives a simple schematic example of two units communicating over a channel. In pactice, a unit will typically be a relatively large component, consisting of tens of thousands of gates in a hardware implementation, *e.g.*a processor with L1 cache, a DRAM controller or a network controller. All communication between units is via messages sent over unidirectional point-to-point inter-unit *channels*, where each channel is buffered to allow units to execute decoupled from each other.

The behavior and abstraction of channels are fundamental cross-platform implementation, composition and debugging of the target system. To these ends, channels are lossless, strictly typed, point-to-point, unidirectional and provide ordered message delivery; in other words channels have the same outward semantics as a standard hardware FIFO. Supporting composition despite unknown delay, given the above channel semantics requires that units be latency insensitive by design. This enables not only compisition of units from idependant developers, but composition across multiple

Figure 1 Basic RAMP Communication Model



platforms.

The RAMP architecture is primarily concerned with defining the interfaces and semantics which are required of the target system in order to maintain the goals of RAMP (see section 3). This in turn will suggest the constraints on the underlying host implementation (see section 4). As the separate sections in this document would suggest, we will maintain a very strict separation between the target and host systems, in order to ensure that RAMP target designs will be portable across host implementations.

3 Target Model

This section describes the target level components of the RAMP architecture and defines their interaction. At the target level a RAMP design is composed of units communicating over channels, by sending messages as shown in figure 2. This section expands on the brief description in section 2 including a discussion of units (section 3.2), channels (section 3.3) and the details of their interaction (section 3.4).

3.1 Introduction to Time

RAMP is designed to support a wide range of accuracy with respect to timing, from cycle accurate simulations to purely functional emulations. Purely functional emulations of course represent the simple case, where no measurement of time is required, and any which exist are incidental. However because a RAMP simulation may require cycle accurate results, an implementation must maintain a strict notion of time with respect to the target system. Thus we introduce the term *target cycle* to describe a unit of time in the target system.

In order to take into account semi-synchronous systems we have defined a unit as a single clock domain. This means that the target clock rate of a unit is the rate at which it runs relative to the target design. For example, the CPUs will usually have the highest target clock rate and all the other units will have some rational divisor of the target CPU clock rate (e.g., the L2 cache might run at half the CPU clock rate). This implies that two units at each end of a channel can have different target clock rates, further complicating cycle accurate simulation.

Of course units are only synchronized via the point-to-point channels. The basic principle is that a unit cannot advance by a target clock cycle until it has received a target clock cycle's worth of activity on each input channel and the output channels are ready to receive another target cycle's worth of activity. This scheme forms a distributed concurrent event simulator, where the buffering in the channels allows units to run at varying target and host rates while remaining logically synchronized in terms of target clock cycles.

The role that target cycles play in synchronization of units is described further in section 3.4. As final note, time in the target system is purely virtual, and thus is not tightly coupled to either real time or the host system's notion of time. The primary goal of the RAMP Design Framework is to support research through system emulation, not to build production computing systems. This distinction is particularly important for hardware (FPGA) host implementations: the goal is not to build computers from FPGAs.

In a target design which is designed to perform a cycle accurate hardware simulation, the target cycle of course will naturally correspond to a clock cycle in an equivalent non-RAMP implementation, however, in software time is a much more complex subject. Having introduced the term *target cycle* and, we now defer a more detailed discussion of time to the following sections, where we will clearly describe what can and must take place within a target cycle.

3.2 Unit Interface

Figure 3 shows a schematic of the interfaces a RAMP unit must support, for an example unit with two input ports (A & B), and one output port (C).

Figure 2 RAMP Target Model

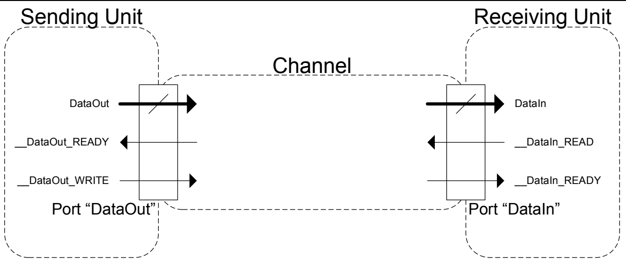
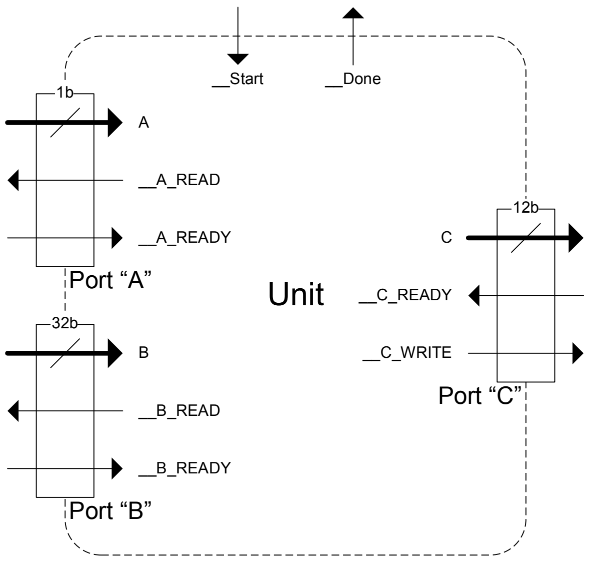


Figure 3 Target level Unit Interface



Ports comprise the input and output interfaces between units and each port is connected to another via a channel. There are two non-obvious points of interest in this figure. First, in addition to the ports, there are two connections labeled `__Start` and `__Done`, which are used to trigger the unit to perform one target cycle worth of action. Second, the ports are each given a message size, or bitwidth. In general, RDL supports more complex messages such as *structured messages*, or many types of messages through the use of *union messages*.

As the figure shows, each port has a FIFO style interface, which provides a natural match to the channel semantics as described in detail in section 3.3. Input messages are consumed by asserting the appropriate `__Xxx_READ` when the associated `__Xxx_READY` is asserted. Similarly output messages are produced by asserting `__Xxx_WRITE`, when the associated `__Xxx_READY` is asserted.

It should be noted that while the above description referred to "signals," which can be "asserted," these constructs can just as easily be represented in software. In software, `__Start` and `__Done` might be represented by a synchronous function call (`__Start()`), which returns when the unit has finished one target cycle worth of action. The ports might then be represented in an object oriented fashion, the same way that a normal FIFO would be in Java or C++. Note again that we do not suggest any connection between target cycles and any unit of time in the host implementation, a fundamental decoupling which should be clear from the description of `__Start()`as a synchronous function call. For descriptions of both hardware and software implementations see section 5.

For reasons that will be made clear in section 4 we use the term *inside edge* to refer to the interface shown in figure 3; the collection of the various ports and the two control signals. The basic goal of this interface is to decouple the implementation of the unit (a complex and time consuming task requiring a researcher to write verilog, java or similar code) not only from the host, but from the rest of the target system as much as possible. Currently a complete decoupling is impossible due to the limitaton that the number and types of the ports (see section 3.3) must be staticly assigned to each unit at design time. However it is our hope that more complete parameterization, polymorphism and optional port connections will, in the future, decrease this design inflexibility.

Given the above goals and limitations, we describe here the functional operation of the inside edge interface: the interface between each unit and the rest of the RAMP combined target and host system. This description is written in terms of a hardware implementation for conciseness and clarity, not because of any fundamental bias in RAMP. Note also that we use here the term "message" which will be more formally defined in section 3.3. In hardware the following interaction will occur between a unit and an external entity refered to as a wrapper (see section 4.1):

1. Before each target cycle the wrapper will present each port with either zero or one mes-

sages, signalled by asserting or deasserting `__Xxx_READY`. This is a key point, as it implies that all messages for a particualar target cycle are delivered atomically before that cycle can begin. Furthermore each message is delivered atomically, never in pieces.

2. The wrapper will signal the unit to start a target cycle by asserting `__Start`.

3. The wrapper will wait for the unit to signal that it has completed the target cycle by asserting `__Done`. Note that in software the start/done signalling may be a synchronous function call.

4. The wrapper will accept exactly zero or one messages from each output port for which the unit asserted `__Xxx_WRITE` at any point in the target cycle. The wrapper will only accept messages from ports where the `__Xxx_READY`was asserted during this target cycle. Any attempt to send messages over unready ports will result in the loss of said mesages. Any message accepted must be delivered in order, in accordance with the channel model, as described below. Again, messages are accepted atomically.

3.3 Channel Model

The key to inter-unit communication, as well as many of the fundamental goals of the RAMP project, lies in the channel model. In addition to the inside edge of the wrapper, the channel model is the other main piece of the target model.

The channel model can be quickly summarized as lossless, strictly typed, point-to-point, and unidirectional with ordered delivery. This should be intuitively viewed as being similar to a FIFO or circular queue with a single input and output, which carries strictly typed messages. In fact, these example constructs will often be the building blocks of channel implementations. From this quick outline we now build upon the basic channel model by describing how they are strictly typed, and their full behavior as a component of a target system.

Channels are strictly typed with respect to the messages they can convey. A *message* in RAMP is the unit of data which a channel carries between units, however, this does not in any way restrict the use or movement of data within a unit. In keeping with the flexibility goal of RAMP, and to expand it's utility as a performance simulation platform, we also introduce the concept of a message *fragment* to describe the unit of data which a channel carries during one target cycle.

Figure 4 illustrates the difference between a message and fragment. The channel (represented as a concatenation of registers and a FIFO for reasons which will be clear shortly) carries exactly zero or one 8bit fragments on each target cycle. The units, however wish to communicate using 40bit messages. Therefore the messages must be split into 8bit fragments for transport over the channel at a rate of one fragment per target cycle. This means that the sending unit may send at MOST one 40bit message every five target cycles. To enforce this limit, the `__Xxx_READY` signal in the sending unit will only be asserted one out of five times `__Start` is asserted, for a 20% duty cycle.

Of course the inverse example is equally valid: a message may be smaller than the fragment size of the channel. In this case a message may be sent on every target cycle, however bear in mind that a channel will carry exactly zero or one fragments per target cycle. This means that the channel may carry no more than a single message per target cycle.

Thus far we have explained the interaction of messages, fragments and channels. Now we justify the use of this rather complex model of communication.

Fragments provide RAMP with a great deal of flexibility in the definition and performance characteristics of channels. Fragmentation allows RAMP to decouple the size of messages, which is a characteristic of a unit port, from the size of data moving through the channels. This allows channels to be parametrized with respect to key performance parameters without sacraficing interoperability. Additionally, the concept of fragments is intricately tied to the notion of target cycles. Just as a target cycle is the unit of time in RAMP, the fragment is the unit of data transfered over a channel per target cycle. Further discussion of the interaction between time and channels is deferred to section 3.4.

There are three primary variables associated with every channel: bitwidth, latency and buffering, as illustrated in figure 5. It should be immediately clear that the bitwidth of a channel (the number of bits it can carry per target cycle) is exactly equal to the size of a fragment. Latency, of course, is the minimum number of target cycles which a fragment must take to traverse the channel. However that the maximum number of cycles a fragment may reside in a channel before being accepted by a unit is not known, and may vary according to the run-time behavior of the unit. This is the key reason units must be designed in a latency insensitive manner.

The final primary channel parameter buffering, is then defined as the number of fragments which the sender may send before receiving any acknowl-

Figure 4 Message Fragmentation & Target Cycles

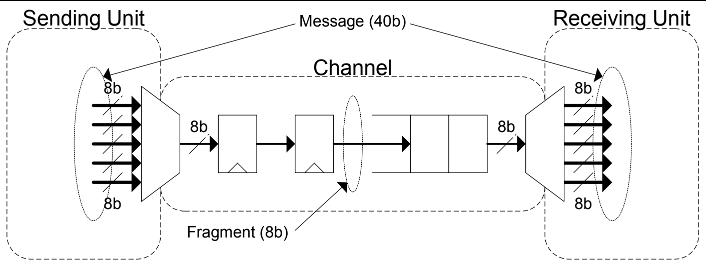
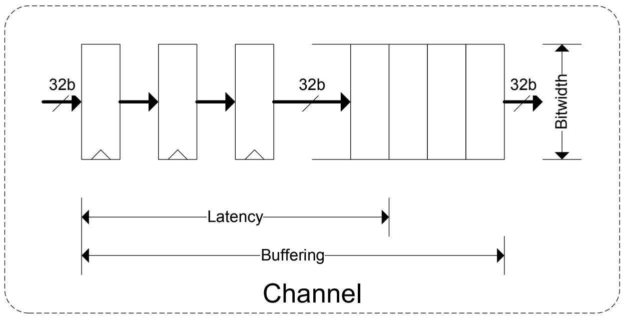


Figure 5 Channel Model with Parameters



edgement of reception from the receiver. In general a channel which must support maximum bandwidth communication will require $buffering \geq 2 * latency$. However, it is easy to imagine a channel which in fact does not need to be capable of bandwidth equal to its bitwidth. In this case, the buffering may actually be less than 0, indicating that data cannot be buffered within the pipeline section of channel shown in figure 5.

Of course to be useful a channel must have a minimum bitwidth of 1. However in order to ensure that the RAMP architecture can be feasibly implemented and maintain a complete decoupling between units, both the latency and buffering must be at least 1. The minimum latency of 1 simply states that the receiving unit cannot receive a message in the same cycle that the sender sends it. This is required in order to ensure that all messages may be presented to a unit at the begining of a target cycle, while it may send a message at any point during the target cycle. The latency requirement ensures that all data dependencies between units are separated by at least one target cycle.

The minimum buffering requirement of 1, exists for a similar reason: without this minimum buffering two units connected by a channel will have a control depedency of zero cycles. This is because the sending unit's ability to send a message on a certain cycle will depending directly on whether or not the receiver receives a messages on that same cycle.

With these two minimum requirements, a natural and efficient implementation of the handshaking is a credit based flow control. What's more credit based flow control will happily tolerate the fact that latency must be accounted for both in the data transfer (fragments moving forward) and the handshaking (credits moving backward). At startup, the sending unit would be given a number of credits equal to the buffering capacity of the channel, thereby allowing it to send that many fragments prior to the receipt of any additional credits. Of course the receiver should return credits to the sender, as it consumes fragments, thereby freeing buffer space. Because it will take *latency* cycles for the fragments to reach the receiver and another *latency* cycles for the new credits to reach the sender, the channel will require $buffering \geq 2 * latency$ to achieve $bandwidth = bitwidth$.

An alternative implementation would be a distributed FIFO where stage would be seperated by one target cycle of latency, both for data travelling forward and flow control travelling backwards. Because both flow control and data are subject to latency, each stage will require two fragments worth of buffering, providing for the same $buffering \geq 2 * latency$ required to reach $bandwidth = bitwidth$. This introduces the fourth channel parameter, which will normally be of secondary interest: the backwards latency of a channel. Providing a separate control on forwards and reverse latency greatly increases the utility of the timing model, and allows RDL to describe a wide variety of models, including certain asynchronous circuits.

In general, the benefit of enforcing a standard channel-based communication strategy between units is that many features can be provided automatically. Users can vary the latency, bandwidth, and buffering on each channel at configuration time. The RDL compiler also provides the option to have channels run as fast as the underlying physical hardware will allow to support fast functional-only emulation. We are also exploring the option of allowing these parameters to be changed dynamically at target system boot time to avoid re-running the FPGA tools when varying parameters for performance studies.

The RDL compiler will build in support to allow channels to be tapped and controlled to provide monitoring and debugging facilities. For example, by controlling the start and done signals, a unit can be single stepped. Using a separate automatically-inserted debugging network, invisible to target design, messages can be inserted and read out from the channels entering and leaving any unit.

In this section we have described the channel model and the difference between fragments and messages. We have also described in detail the parameters of the channel model, and their interaction with the flow control scheme.

3.4 Unit-Channel Interaction with Time

Up to this point we have given a broad description of time, units and channels in a RAMP target system. We have even gone so far as to describe, in terms of a possible hardware implementation, the semantics of the inside edge. In this section we discuss the composability of these individual components and their system wide interaction, especially with respect to time.

On each target cycle, the channel will carry exactly zero or one fragments. This restriction is the key to advancing target cycles in a cycle accurate RAMP simulation, in a functional emulation this is of course a moot point. Time, at the unit level, is advanced upon the reception of a fragment over each channel, which of course, will make zero or one messages availible on each input port. In order to advance time in the absence of a message being sent over a channel, the channel will in essence carry an "idle fragment."

We have very carefully *not* given first class status to the concept "idle fragment;" it is not a formal RAMP term, because there is no actual requirement that idle fragments exist. Despite the fact that idle fragments may or may not exist, they are a convenient logical construct to explain the mechanism whereby target time advances in the absence of messages. Using this logical abstraction, it is possible to explain how time advances and is synchronized in a hypothetical RAMP target system. When the target system is started, each channel might be filled with a number of idle fragments equal to it's latency. Thereafter on each target cycle, the same port mechanism which provides multiplexing/marshalling of messages into fragments will generate either a real message fragment or an idle fragment if there is space available in the channel. The primary reason we do not give first class status to idle tokens is that we can easily imagine situations in which they are not nessecary, such as when a channel is directly implemented as registers and a FIFO, or they would be too expensive to send.

It is important to point out that back-pressure is an important part of the channel model. Because units can chose whether or not to consume a message on each target cycle, it is possible for a channel to become full. This becomes important as, on each subsequent target cycle the sending unit will not be able to produce a new message. Yet the sending unit will still be told to advance by a target cycle, allowing for example a non-blocking router unit.

Also of key important is the fact that units must not in any way be sensitive to target cycles delays which are external to their own implementation. This is a restriction on RAMP designs; the units must be latency insensitive. The fundamental reason for this requirement is derived from the goal of RAMP to support cycle accurate performance simulations without requiring changes to the functional implementation. The idea here is that a RAMP system can be configured, by changing the parameters of the channels (bitwidth, latency, buffering), to simulate a wide performance space. In addition this ensures that any one unit can be replaced with a functionally identical one, allowing for the painless performance testing of a new architectural component or implementation.

This restriction, while key to large scale systems design, presents a major drawback of the RAMP Design Framework for certain low level projects, many of which aim to use RDL as an implementation, rather than emulation, language. As such we are working to provide a set of primitives to describe the existence of cycle and timing dependencies at a very coarse level. In point of fact, if one designer takes responsibility for creating a collection of latency sensitive units, the target system will function accurately, however with the disadvantage that this will heavily complicate compatibility and retard performance research.

3.5 The Target Model

In the above sections we have discussed the complete model of a RAMP design at the target level. We have discussed units in terms of their interface, the inside edge, which is composed of a number of ports and certain emulation support signals. We have also discussed channels, their properties and parameters and the difference between messages, the unit of transfer between units, and fragments, the unit of transfer over channels. The discussion concluded with the details of the interactions between units and channels in terms of the progression of target level time, measured in target cycles, and the latency insensitive design requirement.

This section is deliberately abstract, with the explicit intent of being vague about implementation details. This is done to ensure that RAMP has no platform or language bias, and can in fact manage emulations or tests including several platforms connected together. The next section will discuss the details of implementation, including the requirements RAMP makes of an prospective implementation platform, and the interactions between implementation and abstraction.

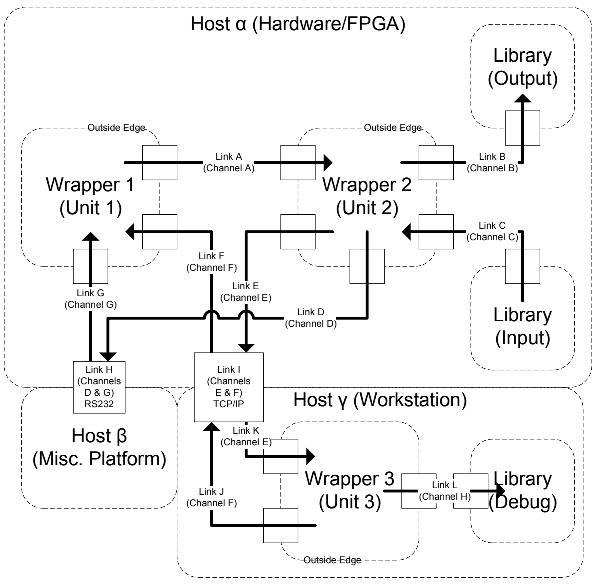
4 Host Model

In comparison to the crisp, rich target model in section 3, the host model if RAMP is far more sparse, with a much smaller glossary. Of course this is one of the goals of RAMP: to avoid overspecifying implementation.

We will also define most of the concepts at the host level in terms of the target level concepts which motivate their existence and parameters. This is done because most host level concepts exist solely to define those requirements imposed on a RAMP implementation by the target model. Time at the host level is measured in host cycles, units are encapsulated in wrappers (section 4.1), and channels are implemented over links (section 4.2). The final construct, which has no analog in the target system is that of the *engine* (section 4.3) which drives the emulation.

Figure 6 shows the eventual implementation goal of RAMP. In this section we will define and clarify the constructs needed to realize this lofty goal while supporting the target model.

Figure 6 RAMP Host Model



4.1 Wrappers

Figure 7 is the direct expansion of figure 3 to include the wrapper, links and control mechanisms required to implement the pure target model of section 3. The key point of interest in figure 7 is the wrapper around the unit, which was briefly mentioned in section 3.2. The wrapper is the container for all of the implementation details required to implement the inside edge interface.

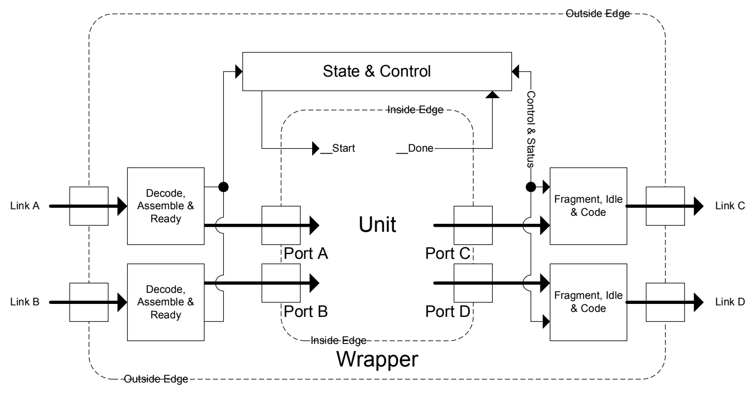
We can now also introduce the term *outside edge* to describe the interface between the wrapper and the links, as well as the rest of the host implementation. The fundamental job of the wrapper is to construct and support the insidge edge interface and translate it to the outside edge interface. This includes the following functionality:

1. Multiplexing/marshalling of messages down to fragments. In fact the wrapper is also responsible for converting from whatever data representation is supported by the target level, to whatever representation is required to actually transport data over links.

2. Demultiplexing/demarshalling of fragments to messages. This is the inverse of above.

3. Target cycle firing. This includes the logic described in section 3.2 and 3.4 for determining when a target cycle may begin based on incoming fragments (including idles) and generating whatever implementation of "idle fragments" is appropriate.

Figure 7 Wrapper around a Unit



4. Enforcement of channel semantics, including ensuring that no more than one message is read or written through each port on a given target cycle. Also ensuring that message transmission and reception is atomic, despite the fact that message delivery, because of fragmentation, is not.

To this end, the wrapper will need to contain a significant amount of functionality. In addition to the link interfacing logic, which is described below, the wrapper must include state and control logic required to manage target cycle firing in the face of input links which may be temporarily empty or output links which may be full. Of course, the firing conditions will also depend on the automatic debugging functionality, which in many cases will include the ability to pause or single step a unit. The main rules for firing have been outlined in section 3.2.

In general because wrappers are conceptually simple, but may involve strange parameter variations, they will be automatically generated by the RDL compiler as described in section 5. This is possible because wrappers are implementation lanaguage dependant, but require no design information beyond basic link parameters, and what would be part of the target model.

4.2 Links

Links are the host level analog of channels. Unlike channels though, RAMP imposes almost no restrictions on links, other than their ability to support the channel model described in section 3.3, thus links can be implemented over nearly any data transport technology. Examples include direct hardware implementation of registers and FIFOs, software circular buffers, high speed serial links, busses, or even UDP/IP packets. Of course not all of these implementations lend themselves to direct channel implementation, for example networks such as Link I in figure 4 are often lossy, and unordered. Thankfully protocols such as TCP can provide the guarentees required.

Unlike wrappers which can be generated automatically, links will often include functionality which is highly implementation platform as well as language dependant. Thus while some links (such as circular buffers in software and FIFOs in hardware) will be automatically generated, others like TCP/IP connections will require pre-written library components to provide their functionality. The RDL compiler includes a number of ways to easily create and extend these libraries.

Because of the relative lack of restrictions on links, there is suprisingly little about them which must be defined. For details about the implementation and composability of links we refer the reader to section 5 wherein we discuss example implementations.

4.3 Engine

In addition to units and links, most implementations of a RAMP design will require some way to drive them, *e.g.*a clock in hardware or a scheduler in software. The exact job of an engine is to decide when and which wrappers are allowed to run, often in conjunction with the wrappers themselves.

In hardware, this task often reduces to providing the reset and clock signals required for a simple synchronous design, thus keeping the engine simple. In a hardware simulation of course the engine might use behavioral constructs to generate a clock signal out of thin air, as it were.

In software the engine is effectively a user level thread scheduling package, where each wrapper (unit) represents a single thread, and the engine must decide which ones to run. RDL and the compiler both admit the possibility of a wire variety of schedulers each with different policies. This has led to a recent interest in supporting efficient software emulations of such projects as Click[] and P2[].

5 Implementation

In the previous sections, 3 and 4 we described in detail the conceptual models associated with both the target and host RAMP systems. However, aside from providing examples where nessecary using hardware terminology, we have avoided implementation details as much as possible. The primary reason for this is to clearly separate the theoretical basis of RAMP from the practical implementation.

This section describes in more detail our first cut at implementing not a specific RAMP design, but the tools required for constructing RAMP designs. We discuss a new language tailored to RAMP, called RDL, it's conversion to Verilog, Java and C and the structure of the compiler.

5.1 RDL

In order to provide a high level description of the various target systems and units that RAMP will be called upon to work with, we have developed the RAMP Description Language or RDL (pronounced "riddle"). First and foremost, RAMP is clearly targetted at large systems where components are created and implemented separately. As such, a large part of the features in RDL were motivated by the need to tie disparate designs together, in a simple, controllable fashion. RDL therefore is best described as a hierarchical declarating programming language, and a possible subset of a future languages.

RDL is a combination of a structural modelling language (similar to the Liberty[] language developed at Princeton) and a simple netlisting language, such as a subset of Verilog or VHDL might provide. At the current time RDL includes support for hierarchical namespaces, messages (both simple, structured and union) and units (both leaf and hierarchical). RDL does not, and will never, provide for the specification of leaf unit behavior, as the RAMP project is aimed to tie together existing designs.

Unit designers must produce the RTL code of each unit in their chosen hardware design language or RTL generation framework, and specify the range of message sizes that each input or output channel can carry in RDL. For each supported hardware design language, the RDL compiler automatically generates a unit wrapper that interfaces to the channels and provides target cycle synchronization. In addition the RTL code for the channels is generated automatically by the RDL compiler, from the connections and hierarchy specified in the RDL source.

Program 1 Example RDL showing late binding

```
namespace                       Base
{
    message  bit[0x20]        DWORD;
};
namespace                       Extend
{
    message  bit[1]         ::Base::BIT;
};
namespace                       UseRename
{
    message  ::Base::BIT  LOCALBIT;
};
```

5.1.1 Keywords, Numbers and Identifiers

Program 1 is a very simple fragment of RDL which shows the basic structure of an RDL file. This RDL file consists of three namespace inside of the base design namespace, called "Base," "Extend," and "UseRename." In each namespace there appears a single message declaration, "DWORD," "BIT," and "LOCALBIT." Aside from the basic features of RDL (namely that it is a structural modelling language or a hierarchical netlisting language) program 1 hints at several notable language features which merit discussion.

First and simplest, any and all numbers in RDL may be in base 2, 8, 10 or 16. A number which starts with a digit 1 through 9, is assumed to be in decimal. A number which starts with a 0, must contain a base indicator (b for binary, c for octal, d for decimal or h for hexadecimal) followed by a number in that base. Thus the number ten could be written in a myriad of ways: 10, 0b1010, 0c12, 0d10 or 0xA.

Slightly more complicated than numbers are various identifiers in RDL. Identifiers, as in C, must start with an underscore or a letter, and may contain underscores, letters and numbers and are case sensitive.

In RDL there are two kinds of identifiers: static and dynamic. Static identifiers are so called because they name objects in the static RDL namespace hierarchy, that is objects which appear in RDL source text such as units or platforms. Dynamic identifiers on the other hand name objects in the system described by the RDL text, that is objects in the unit or platform instance hierarchy. Program 1 only contains static identifiers, since there are no actual unit declarations and thus no inputs, outputs or instances. The difference between static and dynamic identifiers will become clear by way of examples in the following sections, for now all that is important is that static identifiers are qualified with "::" and dynamic identifiers with ".".

This simple example also serves to introduce two of the most powerful features of RDL. First, the ability to create namespaces significantly increases the modularity of the language, allowing researchers in one group to create their design independently of another group. Second, RDL allows all constructs, in this example messages, to be named which allows independently created designs to be easliy merged into a single target system. RDL goes so far as to allow complete late and non-local binding of static identifiers. Late binding allows declarations to appear in any order without regard to their use, however non-local binding deserves further explanation. For example, in program 1 the coder has created the **namespace** Base and used a declaration in the **namespace** Extend to extend Base, a perfect example of a non-local binding.

At first this seems a useless feature, why couldn't the coder simply have included BIT in Base to start with? However with the addition of the **include** "Filename.RDL" **as** NewNamespace statement, the ability to late bind names, such as BIT in this example, becomes a powerful mechanism for independant RDL developement. In addition, late binding of declarations allows for algorithmic parameterization, whereby a hierarchicaly defined unit can make use of a component unit which is not declared until later, by another researcher in another file.

The last feature of interest in this sample is the use of qualified identifiers, whereby the coder has used a declaration in **namespace** UseRename to give ::Base::BIT a local name. The details of this statement deserve a minor explanation: a qualfied static identifier is very similar to a file path, navigating through namespaces rather than folders. The "::" qualified in static identifiers is exactly like the "/" qualifier in UNIX path names, both for specifying the root, and relative path segments.

5.1.2 A Simple RDL Example

Shown in program 2 is an example RDL fragment, which we will use to illustrate the basic features of RDL and the compiler. This is a simple RDL description of a 32bit Up/Down counter which will count up by one each time it receives an input message that is an 1 and down by one each time it receives a 0. This counter will produce an output message consisting of it's count value AFTER the appropriate action is taken upon receipt of each input message. Of course if it receives no input, it will produce no output. This can be summed up in a simple state transition diagram such as in figure 8.

Program 2 A 32bit Up/Down Counter in RDL

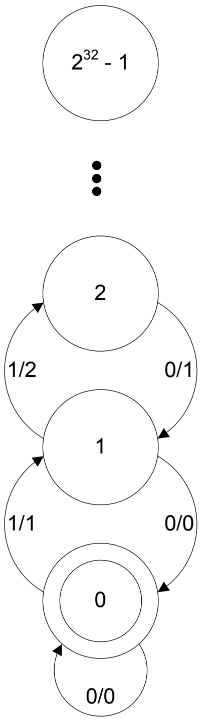
```
unit {
    input bit[1] UpDown;
    output bit[32] Count;
} Counter;
unit {
    instance IO::SwIn UserIn
        (Value(InChannel));
    instance Counter Counter
        (UpDown(InChannel),
         Count(OutChannel));
    instance IO::LEDOut UserOut
        (Value(OutChannel));
    channel fifopipe<1, 1, 1>
        InChannel;
    channel fifopipe<32, 1, 1>
        OutChannel;
} CounterExample;
```

While this is an extremely simplistic example, and much smaller than a typical unit, it illustrates the basics of RDL. The ::Counter is declared to accept unstructured 1-bit messages at its port "UpDown" (::Counter.UpDown) and produce 32-bit messages at its output port "Count" (::Counter.Count). Of course this is a leaf unit, which will be implemented directly in the host language (Verilog, Java for example).

RDL and the compiler also support hierarchically defined units like "CounterExample" in this code fragment. Inside this unit, there are two channels, shown with detailed timing models, which are used to connect the three unit instances. Note the use of named port connections similar to Verilog. Positional and explicit port specification is allowed, including the ability to specify connection of a local channel to a port significantly lower in the hierarchy, without explicit pass-through connections at each level.

RDL also supports declarations for host platforms (eg. an FPGA board or computer with specific I/Os) and mappings of a top-level unit onto a platform. Platform declarations include the language (e.g., Verilog or Java) to generate and the specific facilities available for implementing chan-

Figure 8 Counter State Transition Diagram



nels on the host. The back end of the compiler is easily extensible to support new languages, and new host implementations.

A mapping from a unit to a platform may also include more detailed mappings to specify the exact implemention of each channel. The compiler can take just such a mapping and produce all of the necessary output to instantiate and connect the various leaf units, which have been implemented in the host language.

A full reference for the RAMP description language can be found in appendix B. A more complete example is given in the next section, but we will refer back to this example later, as it is simple enough to be easily understood. In addition the complete source code, and instructions for compiling and mapping to a Xilinx FPGA or a Java host are provided at http://ramp.eecs.berkeley.edu.

5.1.3 Another RDL Example

The counter example in the previous section is clearly overly simple for a RAMP unit. Because RAMP units must be latency insensitive, in general they will be relatively large components of a design, to use the examples from section 2, a processor with L1 cache, a DRAM controller or a network controller. In this section we present yet another RDL example, program 3, which illustrates the declaration of three components: a processor, a cache (presumably L2 or lower) and a memory controller of some kind.

The memory system is declared to deal in bursts of 256 bits or 32 bytes, and is byte addressable with a 32 bit address. As such a burst of data is 256 bits, and the corresponding address is $32-log_2(256/8) =$ 27 bits. A Store then is a structured message containing a BurstData and the BurstAddress at which to write it back. Loads are split-phase in this example with two separate messages: a LoadRequest and a LoadReply which are clearly nothing more than new names for BurstAddress and BurstData respectively.

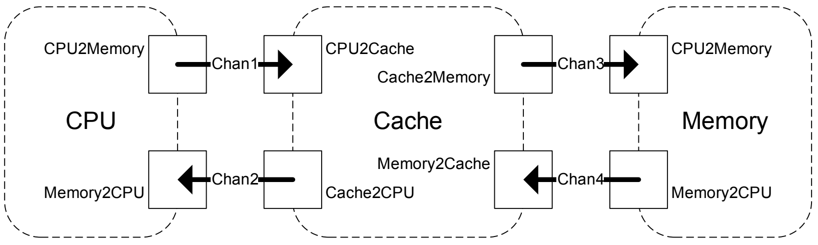
After all of the relevant structuree messages are declared, there are declarations for the two union messages required for this example. The two union types are called MemoryIn and MemoryOut, where MemoryOut is a simple message which can only carry LoadReply messages. The MemoryIn port declaration is only marginally more complicated as it is a union capable of carrying both Load and Store messages, along with a tag indicating which the current message is. Tags in RDL are automatically assigned in a deterministic and repeatable manner, ensuring that they will not change so long as the union message declaration remains unchanged. Tags can also be explicity specified.

Program 4 completes this example by declaring a top level unit System which instantiates the other three units and connects them as shown in figure 9. The program contains three instance declarations, all of which use fully qualified static identifiers for the units which are being instantiated. This was done in order to ensure that they have been correctly named despite the fact that we have no shown the placement of the declaration for System in the namespace hierarchy.

System also contains four channels, connecting the appropriate ports of the instances. The channel declarations inside of System start with the **channel** keyword, followed by a channel model, in this case FIFO1x16, followed by the name of the channel and a pair of dynamic identifiers for the ports it connects. Note that channel names are also dynamic identifiers, but they cannot be qualified. The port names however can be qualified dynaminc identifiers, which may in fact refer to ports on subinstances, allowing easy hierarchical composition without having to pass information through a port at every level of the instance hierarchy.

The last feature of note in this program fragment is the channel model declaration at the top. This declares a channel model named FIFO1x16, which we reference four times inside the System unit declaration. The model is of a 16 deep, 1 bit wide

Figure 9 A Simple Computer System



FIFO, with a 1 cycle latency, as outlined in section 3.3.

5.2 RDL Compiler & Toolflow

In the previous section we gave a number of examples of RDL, and it's many features. The key to translating RDL into a working RAMP design is of course the RDL compiler. However because the RAMP architecture is complicated and cross-implementation with allowances for independant developement of various units, figure 10 deserves brief mention before we delve into the details of the compiler.

Shown in figure 10 is a simple illustration of the RAMP toolflow. Shown at the top are the steps to create a target system in a specific implementation language (Verilog or Java for example). The box in the middle represents a complete RAMP design including all of the information nessecary to generate a complete runnable simulation. The bottom box then represents the steps required to actually produce a runnable simulation, including rerunning the RDL compiler to produce the wrapper and link code, followed by the native host toolflow, be it a JVM for an FPGA system such as BEE2.

5.3 Shell Verilog

In this section we give an example of the inside edge shell Verilog generated by the RDL compiler from program 2. The Counter.RDL file described in section 5.1.2 would be run through the RDL compiler (rdlc) with the command:`java -jar rdlc.jar -shell:‘‘Counter Verilog’’ CounterExample.RDL` resulting in the verilog shown in program 5.

Included in this module is a port declaration for each RAMP port, and a number of control signals, as hinted by figures 2 and 7. Notice that in addition to the `__Start`and `__Done`signals mentioned above, a `__Clock`and `__Reset`are included to support system-wide reset of the the RAMP simulation. The final two declarations are for local parameters (in Verilog these are constants which cannot be overridden at instantiation time) for the bitwidths of the various ports. In a unit with union ports, there would also be local parameters giving names to all the tag values used on the union port.

With this Verilog shell of the inside edge interface in hand a researcher could fill in the functionality required. Later on, an RDL design incorporating this unit would result in the generation of the wrapper, which is responsible for instantiating this unit. By adding the above file, with appropriate functionality, into the synthesis project, a complete RAMP simulation design can easily be produced.

5.4 Shell Java

In this section we will cover the basics of working with a RAMP simulation generated in Java. Java was chosen as the primary software output language of RDLC not because it is well suited to simulations, but because it is easy to read, easy to implement and has a superset of the features of most other software languages. Thus we believe it is an easy matter to write the output code for other software languages by copying and modifying the Java output code.

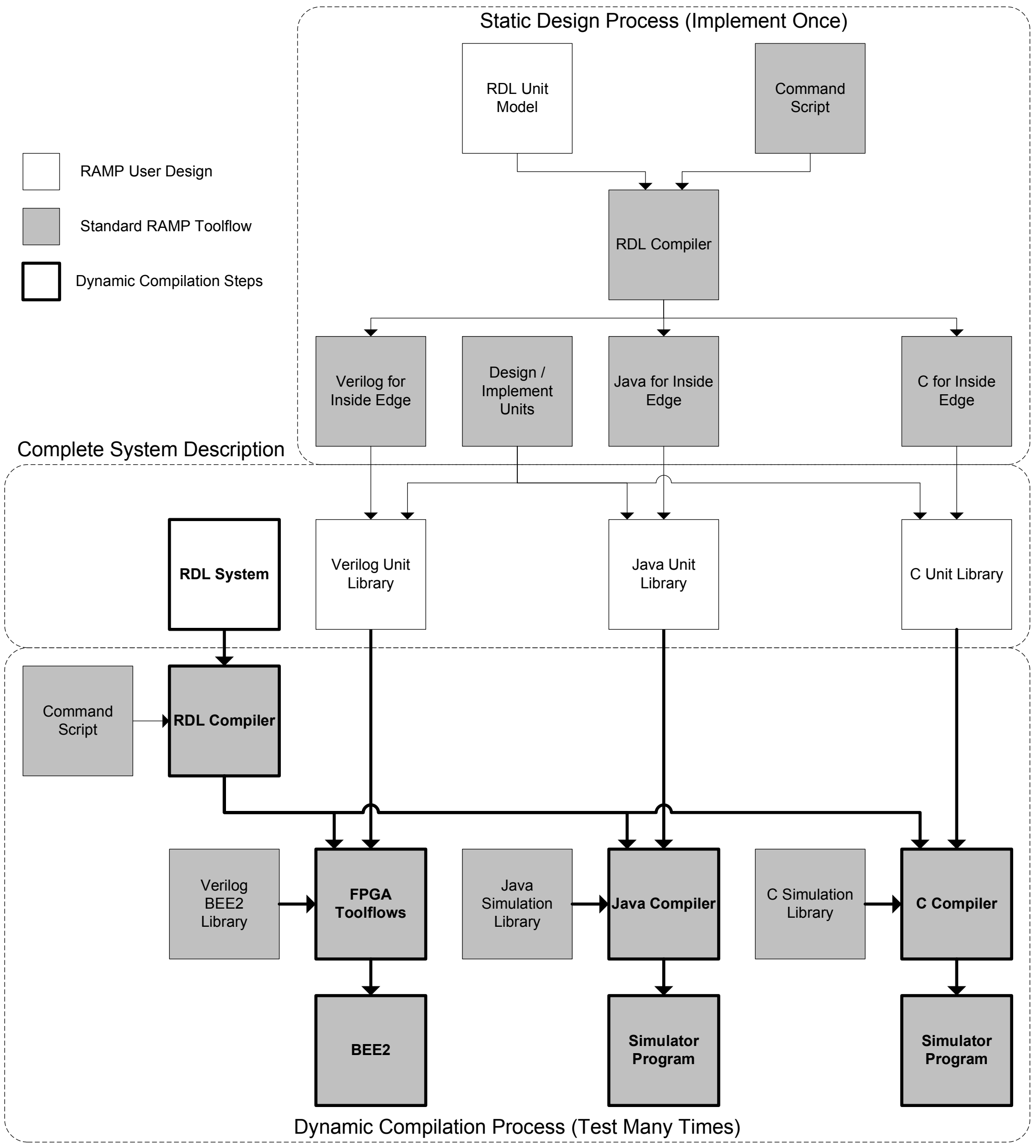
Shown in program 6 is the java inside edge shell generated from program 2. In addition to this one file, the command `java -jar rdlc.jar -shell:‘‘Counter Java’’ CounterExample.RDL` , will generate a series of support files, for all of the classes and interfaces mentioned in this file.

What follows is a list of the relevant statements in the java shell.

Package: There is a package declaration placing this in the root package (JavaShell) of the design. In a larger design, the package hierarchy will exactly mirror the RDL namespace hierarchy.

Class Declaration: We are declaring a class named Counter, which implements the standard RAMP unit interface. Notice that the RAMP unit interface is named as

Figure 10 RAMP Toolflow



Program 6 Java Shell for Counter.RDL

```
package JavaShell;
/**
 * Class:   Counter
 * Desc:    TODO
 * @author gdgib
 */
class Counter implements ramp.library.__Unit {
    protected ramp.library.__Input<ramp.messages.Message_1> UpDown;
    protected ramp.library.__Output<ramp.messages.Message_32> Count;

    /**
     * @see ramp.library.__Unit#Reset()
     */
    public void Reset() {
    }

    /**
     * @see ramp.library.__Unit#Start()
     */
    public boolean Start() {
    }
}
```

JavaShell.ramp.Unit indicating is is the interface Unit in the ramp package, which will contain a number of other support classes and interfaces.

Ports: There are declarations for an input and an output port, as represented by the JavaShell.ramp.Input and JavaShell.ramp.Output interfaces. These interfaces take advantage of the class specialization availible in Java 1.5, to use the Java typechecker to ensure that only the appropriate message types can be sent or received on these ports.

Methods: There are empty method implementations for the two methods inherited from the JavaShell.ramp.Unit interface, along with JavaDoc references to the original two methods.

In this section we have summarized the basic features of the Java inside edge shells, as generated by the RDL compiler.

5.5 Mapping to Platforms

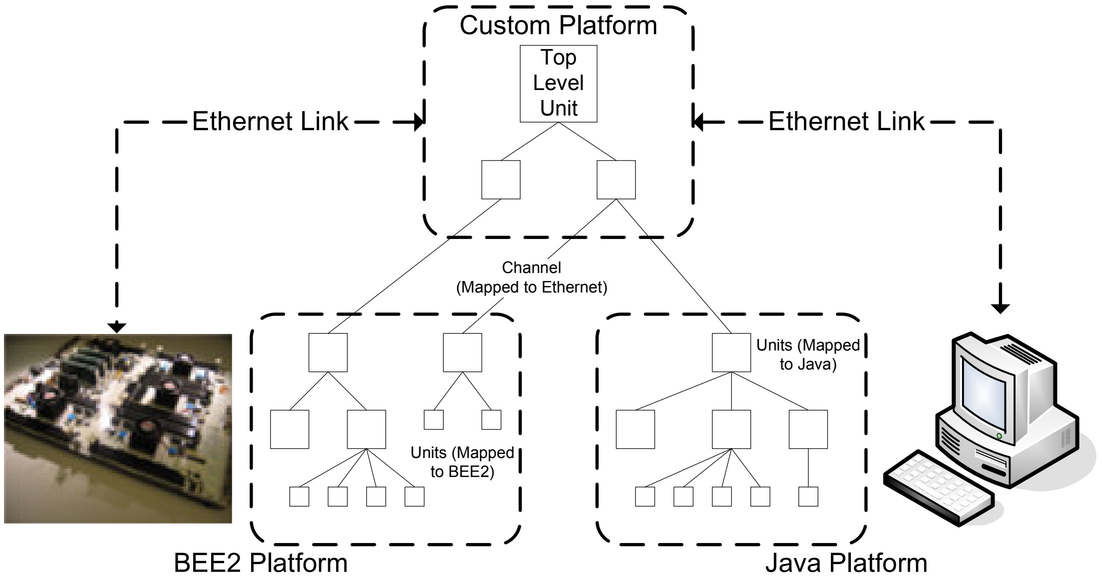
In order to support designs which can or should span multiple platforms, RDL includes cosntructs to describe both host platforms and the mapping from a hierarchical target design to these platforms. First off, RDL acknowledges that most host systems will be comprised of multiple platforms. This is manifest in the hierarchical construction of platforms, as shown in figure 11 where a workstation platform is composed of a BEE2 and a desktop computer connected by Ethernet.

In addition to the requirements imposed by real platforms (*e.g.*that links may not be point to point like channels) platform specification is complicated by the desire to reduce compilation cycles on very large systems. For example a large design filling a Xilinx V2Pro 70 FPGA on the BEE2 may take many hours to place and route, forcing the RDL compiler to support some form of compile-once, run-many to reduce overall target implementation costs.

In order to cope with these requirements RDL provides for creating a hierarchy of units, a hierarchy of platforms and a hierarchy of mappings which group one or more units onto a specific platform. While on the surface a map groups units onto a platform, it must also provide for specifying which unit subinstances are handled by it's map subinstances, and which channels are carried by which links.

While this requires a good deal of specification on the part of the RDL writer, schemes which place a higher burden on the RDL compiler are infeasible in the short term. As it is, the compiler cannot auto-

Figure 11 Mapping to Platforms



matically partition a design, or cope with multi-hop channels to link mappings. Even without these features, RDL provides an excellent framework for research in these areas by providing a common specification language, allowing a partitioning or network embedding tool to handle the complex algorithms, and leave the implementation to RDLC. We expect this to be an active area of future research and hope to provide a solution in the future, but for now a working RDL compiler is more valuable.

5.6 RDL Compiler Structure

This section describes in some detail the internal structure of the RDL compiler. Because the RAMP architecture is designed to be cross-platform, we have found the structure of the compiler and tools to be vitally important when porting RAMP to a new host implementation. For this reason the compiler is highly modular, with very full abstractions and generalizations wherever possible.

Also, to ease the porting process, and to speed initial developement, the RDL compiler is currently written in Java, as java has a higher functionality-per-line density than many languages and can run on many platforms. We envision a future rewrite of the compiler in C, C++ or a similarly low level language for performance reasons, as java can be slow. However, we believe this will not be nessecary or useful for some time, as ease of modification will be significantly more important during the early stages of the RAMP project.

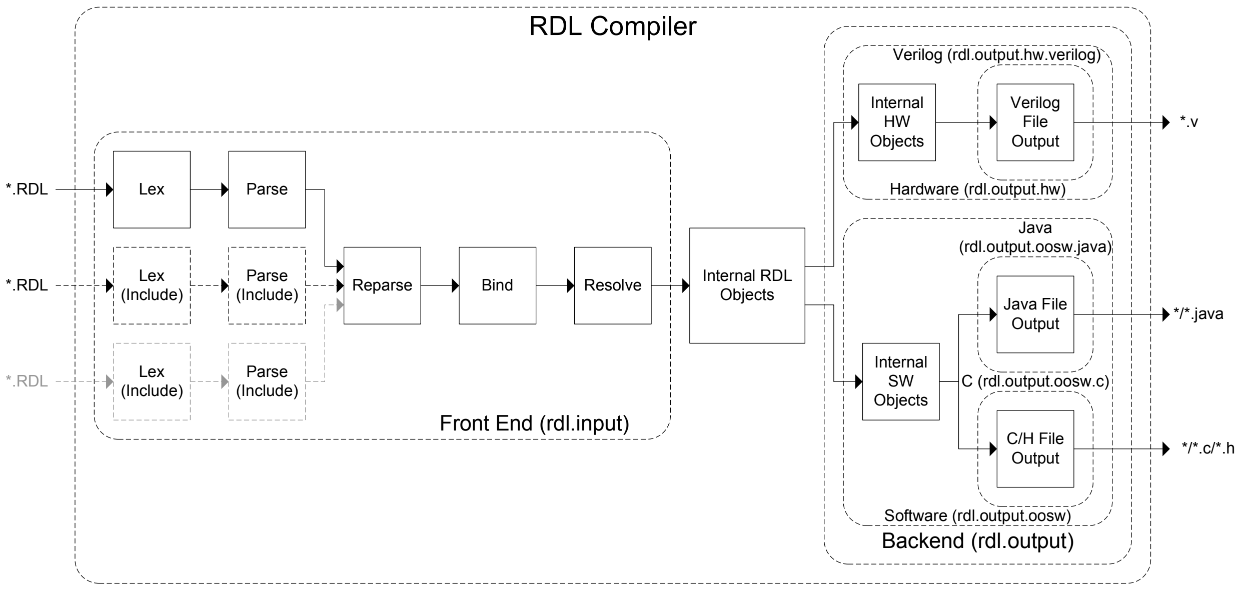
5.6.1 Back End

Figure 12 gives a high level view of the structure of the RDL compiler, with respect to the java packages of which it is composed and the functionality they contain. In our attempts to make RAMP entirely platform agnostic, the RDL read-in and host language generation have been completely separated, and encapsulated as much as possible. Since this is still fundamentally a compiler for the RAMP description language, each output generation package (Verilog, Java and C in figure 12) is expected to take an abstract set of in memory RDL objects and construct the appropriate output.

In order to simplify each output package in light of the fact that most of them really differ only in the text of the generated output, we have created object-oriented software and hardware packages (rdl.output.oosw and rdl.output.hw respectively). These packages take care of transforming the in memory RDL objects into the appropriate structure. From the object-oriented software representation to both Java and C is trivial, as is generating Verilog from the hardware representation. With these intermediate representations adding for example C++ and VHDL as output languages should be trivial.

There might also be situations where there are two output packages that generate the same language, but for different simulation platforms. For example: RDLC will often be used to generate simulations which must be plugged into existing frameworks with their own different standards for communication, naming, typing and timing, but which

Figure 12 RDL Compiler Structure



are implemented in the same language. In this case there might be two seperate C output packages which generate textually different code to make it compatible with existing simulation frameworks.

For more information about the structure of the compiler, please see the RDLC javadocs.

5.6.2 Front End

In the same vein as the output packages, the input package in the RDL compiler is written as a series of java classes, one representing each RDL construct. For example there are classes for units, channels, messages and ports. The RDLC front end as shown in figure 12 consists of all of the code required to instantiate and connect all of these classes to create a composite in-memory data structure representing the RDL design.

In the remainder of this section we give a loose BNF model of this memory structure. Because this memory structure exactly mirrors the RAMP description language, this may also be taken to be a loose BNF model of the constructs in RDL. However it is NOT a BNF grammar of RDL itself, that is presented in appendix B.

$Design ::= Namespace$

$Namespace ::= Element*$

$Element$
  $::= Namespace$
  $|\ Channel$
  $|\ Message$
  $|\ Unit$
  $|\ Platform$
  $|\ Map$

$Channel$
  $::= FIFO$
  $|\ FIFOPipe$

$Message$ ::=
  $::= Simple\ Message$
  $|Structured\ Message$
  $|Union\ Message$

$Unit ::= UnitInstance$

$UnitInstance ::= UnitField*$

$UnitField$
  $::=$ **input** $Port$
  $|$ **output** $Port$
  $|UnitInstance(\ Channel*\ )$
  $|Channel\ Source\ Destination$

$Platform ::= PlatformInstance$

$PlatformInstance ::= PlatformField*$

$PlatformField$
  $::= PlatformInstance$
  $|Engine$
  $|Language$
  $|Default\ LinkType$
  $|Port\ LinkType$

Program 3 A CPU and Memory Model in RDL

```
namespace Memory
{
    message bit[256]  BurstData;
    message bit[27]   BurstAddress;
    message struct
    {
        BurstAddress  Address;
        BurstData     Data;
    }                 Store;
    message BurstAddress  LoadRequest;
    message BurstData     LoadReply;

    message union
    {
        LoadRequest  Load;
        Store        Store;
    }                MemoryIn;
    message LoadReply   MemoryOut;

    unit
    {
        input   MemoryIn   CPU2Memory;
        output  MemoryOut  Memory2CPU;
    }                      Memory;
};
namespace CPU
{
    unit
    {
        output  ::Memory::MemoryIn
                CPU2Memory;
        input   ::Memory::MemoryOut
                Memory2CPU;
    } CPU;

    unit
    {
        output  ::Memory::MemoryIn
                Cache2Memory;
        input   ::Memory::MemoryOut
                Memory2Cache;

        input   ::Memory::MemoryIn
                CPU2Cache;
        output  ::Memory::MemoryOut
                Cache2CPU;
    } Cache;
};
```

Program 4 A Simple Computer System

```
channel fifopipe<1,1,15> FIFO1x16;

unit
{
    instance ::CPU::CPU CPU;
    instance ::CPU::Cache Cache;
    instance ::Memory::Memory Memory;

    channel FIFO1x16 Chan1
    { CPU.CPU2Memory ->
      Cache.CPU2Cache };
    channel FIFO1x16 Chan2
    { Cache.Cache2CPU ->
      CPU.Memory2CPU };
    channel FIFO1x16 Chan3
    { Cache.Cache2Memory ->
      Memory.CPU2Memory };
    channel FIFO1x16 Chan4
    { Memory.Memory2CPU ->
      Cache.Memory2Cache };
} System;
```

Program 5 Verilog Shell for Counter.RDL

```
module  Counter(__Clock,
                __Reset,
                __Start,
                __Done,
                __UpDown_READY,
                __UpDown_READ,
                UpDown,
                __Count_READY,
                __Count_WRITE,
                Count);

    input       __Clock, __Reset;
    input       __Start;
    output      __Done;

    input       __UpDown_READY;
    output      __UpDown_READ;
    input       UpDown;

    input       __Count_READY;
    output      __Count_WRITE;
    output  [31:0]  Count;

    localparam  _WIDTH_UPDOWN = 1;
    localparam  _WIDTH_COUNT = 32;
endmodule
```

$Map ::= MapInstance$

$MapInstance ::= MapField*$

$MapField$
$\quad ::= MapInstance$
$\quad |UnitInstance$
$\quad |PlatformInstance$
$\quad |Map\ Unit\ Map$
$\quad |Map\ Platform\ Map$
$\quad |Map\ Channel\ Link$

For more information about the structure and possible ways to extend the RDL compiler, we refer the interested reader to the javadocs, availible from the RAMP website.

6 Project Status

The RDL compiler has been completed and is fully documented with javadocs. The RAMP description language and the RDL compiler are stable, with working examples, and are ready for research use. Timing-accurate simulations have been mostly implemented, but remain untested. The compiler, the counter example, counter example lab and the javadocs are all availible from http://ramp.eecs.berkeley.edu.

7 Future Work

Finishing this document will be a subject of ongoing work, since RDL and RDLC are in active use and still partially under developement, we do not expect this to be a discrete goal. Appendix B needs to be updated for platforms and maps, as do the sections about the RDL compiler above.

Polymorphism, complete integer parameters and generator language constructs are all useful, possible vital extensions to RDL which have not been implemented.

8 Acknowledgements

The authors would like to thank the rest of the RAMP Gateware Group at UC Berkeley, especially Heidi Pan and John Wawrzynek for their suggestions and contributions, in all probability, this document would not exist without their help. We would also like to thank those who took the time to read and comment on this technical report, in particular Dave Patterson and Derek Chiou who commented early and often. Finally we would like to thank Eric Chung, the students of CS252 Spring2006, Ivan Sutherland, those who worked on FLEET and everyone else who worked with or complained about our tools.

A Glossary

Simulation: We use the work simulation to refer to a timing accurate simulation of a target design. This is in contrast to the term emulation, which implies timing inaccuracy.

Emulation: An emulation in RDF implies a timing inaccurate emulation of a target design. Emulation will provide much higher emulator performance, as it removes the overhead of detailed simulation timekeeping.

Host: Host refers to the hardware or software designed or supported by the RAMP project for emulating or simulating a "target" design. This is the actual implemented design. The host model includes the concepts of *wrappers*, *links* and the *outside edge* interface.

Link: A link is a the actual communication facility present in the host system, on top of which a channel is built. Links may be lossy, dynamically routed, have extreme latencies and may not be point-to-point. It is the job of the wrapper in conjunction with the link management code to ultimately handle the complexities of the link and present the unit with an idealized channel through the *inside edge.*

Wrapper: This is the Verilog, Java, C or similar implementation code which forms an interface between a unit, and the exact host system in any given simulation. Wrappers will be dynamically generated by the RDL compiler, though so called "inside edge shells" containing no functionality may be generated to assist in unit development. The wrapper is ultimately responsible for providing a clean set of ports to the unit and hiding from it the details of the host system. As such the wrapper also provides the following control signals (in a hardware host implementation):

- **__Clock:** This is the host system clock.
- **__Reset:** This is the host system reset signal.
- **__Start:** Is a single cycle pulse, driven by the wrapper to tell the unit that it must begin simulating a single target cycle.
- **__Done:** Is a single cycle pulse, drive by the unit to tell the wrapper that it has completed the current target cycle. A unit which can process a target cycle in one host cycle may wire this directly to `__Start`.

Outside Edge: This describes the interface between the wrapper and links as implemented in the host system. The exact details of this interface will likely vary widely with the links the wrapper must connect to.

Engine: The engine is the hardware module or software object responsible for driving the emulation. In hardware, this translates to generating clock and reset signals. In software an engine is tantamount to a user level thread scheduler.

Target: The target is the system which the host is currently simulating. This is the idealized design, in which a RAMP user is interested. The target model includes the concepts of *units*, *channels*, *messages*, *fragments* and the *inside edge.*

Unit: A unit is an indivisible encapsulation of functionality in RAMP which simulates some piece of the target system, in a RAMP compatible fashion (*I.e.*with support for channels, etc). Units should be specialized implementations of target functionality, and should be entirely general from the point of view of the host system, thereby enhacing the composability of the system. Note that a unit will need to be aware of the host semantics of RAMP as described in section 3.4.

Channel: Channels are the abstraction of inter-unit communication in RAMP. Each channel will connect exactly two units, and provide in-order, lossless message transport at the rate of zero or one fragments per cycle. Channels have a number of characteristics:

- **Type:** This includes the widths and types of messages that the channel may carry. This information must match the declared message types the sender and receiver unit are capable of generating and consuming respectively. In RDL this information is automatically generated based on the ports to which the channel connects.
- **Bitwidth:** The width of the channel will be specified in bits. This is the width of the fragments this channel carries, at a rate of zero or one fragments per target cycle. Minimum bitwidth is 1.
- **Latency:** Is measured in target cycles as the minimum transit time of any fragment from the sending to receiving unit. Note that $max(messagesize)/bitwidth$

provides a lower bound on the latency of the channel in any actual implementation. Minimum latency is 1 to ensure that there are no combinational loops.

Buffering: Indicates the number of fragments that the channel can buffer. Minimum buffering is 1 to ensure there are no zero cycle control dependencies. $Bandwidth = bitwidth$ when $buffering \geq 2 * latency$.

Reverse Latency: Is measured in target cycles as the minimum time from a message or fragment being consumed by the receiving unit to the time the sending unit realizes this fact. Minimum latency is 1 to ensure that there are no combinational loops, and by default this will be the same as the latency.

Examples: Expressing the above metrics is tuples $(bitwidth, latency, buffering)$ we can say a 32bit 256 line cut-through FIFO would be $(32, 1, 256)$.

Message: Messages are the basic unit of communication between units. Messages may be structured (composed of smaller messages) or tagged unions of different sub-message types. Messages are the target level unit of flow control.

Fragment: Fragments are the basic unit of data transport over channels. Notice that while channels may carry large messages, they must be broken into smaller fragments at the send and reassembled. See section 3.3 for more details.

Port: A port is simply the name we give to point of connection between a unit and a channel. Port characteristics are entirely static and limited to the type and size of messages the port will carry (which must match the type and size of messages carried by the port it is connected to). In implementation a port will be able to transfer at most one message per cycle and must therefor be as large as the largest message it can support. Ports will operate under FIFO style semantics, with a `__READY`signal to indicate data (on an input) and free space (on an output) along with a `__READ`(input) or `__WRITE`(on an output).

Inside Edge: This marks the interface between the wrapper and the unit. This includes all of the signals associated with the unit's various ports, as well as those listed under *wrapper* above.

B The RDL Language

B.1 BNF Grammar

This section contains the grammar of the RAMP description language in a BNF style format. Text such as **namespace** marks a keyword, which would literally appear in the RDL text. Keywords like this are case sensitive, and should appear in all lower case. Text like *File* marks a BNF variable. Parentheses (and) mark BNF groups, and do not appear in the RDL text, ever. Whitespace is entirely ignored. Comments may be in either standard C/C++/Java style: // to end of line or /**/ blocks.

Numbers can appear in two formats, ones which start with a numeral 1-9 must be in decimal. Numbers which start with a 0, must then contain a base specifier, followed by a number in that base. Valid bases are:

Decimal: 0d...

Binary: 0b...

Hexadecimal: 0x...

Octal: 0c...

What follows from here is a complete BNF description of RDL.

File ::= *Statement*∗

Statement
  ::= *NamespaceDecl* ;
  | *IncludeDecl* ;
  | *ChannelDecl* ;
  | *MessageDecl* ;
  | *UnitDecl* ;

IncludeDecl ::=
  include *FileName* **as** *StaticIdentifier_Simple*

NamespaceDecl ::=
  namespace *StaticIdentifier_Simple* { *File* }

ChannelDecl ::=
  channel *ChannelType* *StaticIdentifier*

ChannelType
  ::= *ChannelTypeSpecFIFO*
  | *ChannelTypeSpecFIFOPipe*
  | *StaticIdentifier*

ChannelTypeSpecFIFO ::=
  fifo < *Number*, *Number* >

$ChannelTypeSpecFIFOPipe$ ::=
fifopipe < $Number$, $Number$, $Number$ >

$MessageDecl$ ::=
message $MessageType$ $StaticIdentifier$

$MessageType$
::= $MessageTypeSpecSimple$
|$MessageTypeSpecStruct$
|$MessageTypeSpecUnion$
|$StaticIdentifier$

$MessageTypeSepcSimple$ ::=
bit [$Number$]

$MessageTypeSpecStruct$ ::=
struct { $MessageFieldDecl*$ }

$MessageFieldDecl$ ::=
$MessageType$ $Identifier$ (, $Identifier$)∗ ;

$MessageTypeSpecUnion$ ::=
union { $MessageUnionFieldDecl*$ }

$MessageUnionFieldDecl$ ::=
$MessageType$ $MessageFieldTag$ (, $MessageFieldTag$)∗ ;

$MessageFieldTag$ ::=
$Identifier$ (< $Number$ >)?

$UnitDecl$ ::=
unit $UnitType$ $StaticIdentifier$

$UnitType$
::= $UnitTypeSpec$
|$StaticIdentifier$

$UnitTypeSpec$ ::= { $UnitFieldDecl*$ }

$UnitFieldDecl$
::= **input** $PortType$ $DynIdentifier\_Simple$;
| **output** $PortType$ $DynIdentifier\_Simple$;
| **instance** $UnitType$ $DynIdentifier\_Simple$;
| **channel** $ChannelType$ $DynIdentifier\_Simple$
{ $DynIdentifier$ −> $DynIdentifier$ } ;

$StaticIdentifier$
::= $StaticIdentifier\_Prefix$? $StaticIdentifier\_Simple$
|$StaticIdentifier$ :: $StaticIndetifier\_Simple$

$StaticIdentifier\_Prefix$
::= ::
| :: $Number$ ::

$StaticIdentifier\_Simple$ ::= $Identifier$

$DynIdentifier$
::= $DynIdentifier\_Prefix$? $DynIdentifier\_Simple$
|$DynIdentifier$. $DynIdentifier\_Simple$;

$DynIdentifier\_Prefix$
::= .
| . $Number$.

$DynIdentifier\_Simple$ ::= $Identifier$

$Identifier$ ::= $[a-zA-Z\_][a-zA-Z0-9\_]*$

B.2 Quick Reference

In section B.1 above, we gave a BNF style grammar for the RAMP description language. In this section we give an english language description of RDL, and the constructs in it. We will restrict ourselves primarly to describing how the constructs of RDL map to the concepts of RAMP layed out in section 3. For a more complete discussion of RDL, and the motivation behind it's design, see section 5.1.

namespace: Hierarchical namespaces are the base constructs of RDL. Each file is represented as it's own namespace, which can optionally contain any number of namespaces. Each namespace has a name within it's parent namespace, thereby forming a tree rooted at a single RDL source file. The name of the root namespace is never required in RDL, but is often used in the code generated by RDLC, and can therefor be set as a command line parameter to RDLC (the design name). In addition to other namespaces, namespaces can contain any of the below listed constructs.

include: In order to support independant developement of RAMP designs, the **include** declaration will include the complete text of another RDL file as a child namespace of the namespace in which the include appears. Include is a simple way to take a complete namespace and move it to an external file.

channel: Named channel declarations, as they appear directly in namespaces, contain only the bitwidth, latency and buffering parameters of the channel. A named channel has no type information, but provides a convenient name for a channel type which may later be used.

fifo: A fifo channel models a hardware style FIFO, with variable bitwidth and buffering. The latency is 1, *i.e.*this is not a transparent FIFO.

fifopipe: This provides access to the complete channel model of a pipeline and FIFO combined, as shown in figure 5. This allows control of the bitwidth, latency and buffering independantly. The minimal value for each parameter is 1.

message: Named message declarations provide a way to name both simple and structural messages in a way which enforces typing. Two unnamed messages are considered to type match if they have the same fields and bitwidths, however named messages only match if they refer to the same named message declaration (they may do so indirectly of course). Thus named messages are both for the convenience of human readable, self-documenting code and to enforce strong typing on top of a very simple bit-type model.

bit: Simple messages consist only of a bitwidth specification.

struct: Structural messages declare a complete data structure with named fields, which can themselves be structured. Unlike structs or classes in most software languages however, structured messages also must maintain ordering of their fields, so that marshalling and unmarshalling of structured messages is well defined. The ordering of the fields is left to right, top to bottom as they are read from the RDL file.

union: A union message is one which carries at any given time any one message from it's union set. Which message is being carried is distinguished by a tag value associated with the message. Fixed tag values may be specified in the RDL declaration of the union field. Union fields without explicit tag assignments will be assigned tags in a deterministic, repeatable fashion, as described in section 5.1

unit: Units are by far the most complicated kind of declaration in RAMP. A unit declaration may have any number of fields, including input and output ports, instances of other units and channels which connect instance ports, along with inputs and outputs.

input: An input declaration indicates the existence of an input port with the given port type on the containing unit. Port types may either be named, or specified directly.

output: An output declaration indicates the existence of an output port with the given port type on the containing unit. Port types may either be named, or specified directly.

channel: A channel declaration inside of a unit, unlike a named channel declaration, specifies not only channel model, but also the two ports which that channel connects. The ports may either be inputs or outputs of the containing unit, or ports on any of the instances or subinsstances within this unit.

instance: An instance declaration indicates the existence of a unit instance within the current unit. The containing unit in this case is hierarchically defined, and can be generated completely and automatically by RDLC. An instance may be of a named unit, or of an unnamed locally defined unit.

static identifier: A static identifier is a name for an RDL declaration, be it a namespace, channel model, port, message or unit. Static identifiers can be fully qualified using the namespace separator $::$. A rooted static identifier is one which is prefixed with a single $::$. A static identifier may also include references to the enclosing namespaces by beginning with a $::\ Number\ ::$.

dynamic identifier: A dynamic identifier is a name for an instance or port within a unit. They may be fully qualified, rooted or parent relative the same as static identifiers. However the separator is . (A period).