DesignCon 2006

A Taxonomy for the Electronic System Level (ESL)

Brian Bailey, Brian Bailey Consulting

email: brian_bailey@acm.org

web: http://brianbailey.us

Abstract

When any new technology or methodology comes along, new terms are created and invariably there is little agreement on those terms. This causes confusion and in bad cases can make it impossible for people to effectively talk about the subject. Standards groups often have to establish the terms that they will use so that they know what they are agreeing to as their first order of business. Then there is the solution space that the ESL tools are addressing which is very broad. It all adds up to a very confusing situation. This paper will introduce elements of a taxonomy of terms and model attributes for this emerging market.

Author(s) Biography

Brian Bailey is an industry and management consultant specializing in the functional verification of electronic systems. He has experience implementing verification tools and solutions for companies of all sizes, and has been involved in some of the most significant advances in the field. He has been proclaimed as one of the visionaries of the industry and today spends most of his time helping start-up companies turn their ideas into practical realities. He is active in the standards community, currently chairing a committee within Accellera, has published two books in 2005 and is a popular contributor to the press and conferences. He graduated from Brunel University in England with a 1st class honours degree in Electrical and Electronic Engineering.

It's a Jungle Out There!

It was only a few of years ago that the Electronic System Level (ESL) market was defined by Gary Smith and Daya Nadamuni at Gartner/Dataquest[i]. What started as a fairly crisp definition was soon to be pushed and pulled in many directions and is no longer clear. It did not take long before the marketing departments of the tool developers took notice of this new space and attempted to rope off the areas in which their companies played. Now users are confused because every tool vendor is telling a different story or claiming to be the leader in the field. There is only one way to solve this problem and that is with a set of standard definitions.

Recently, as part of a survey conducted by the author of this paper, along with Grant Martin and Andrew Piziali, for an upcoming book on ESL, the question was posed: What is your definition of ESL? The answers were widespread, but some common elements included the existence of software, degrees of complexity and concurrency. Others stated that it is about function rather than implementation, is a methodology that utilizes multiple levels of abstraction, and involves design space exploration and architectural decisions. One common thread of agreement was that ESL is everything above the RTL level. This includes such concepts as behavioral, architectural, co-design, system design, and executable specification and encompasses languages, methodologies and tools.

There is another significant divide between definitions in what academics define as models of computation. The most significant split is between dataflow and control. Each of these domains is currently addressed by different flows and tool sets. Gartner splits the ESL landscape into three domains: algorithmic, which generally equates to dataflow, processor and memory configurations, which could also be called block level architecture and communications, and finally control logic, which is what would normally be associated with behavioral design.

Simply stated, there is no single ESL flow, no single set of definitions, and thus a plethora of tool companies attacking different pieces of the problem, each of which contributes to filling a part of the total space called ESL.

For the purposes of this paper, the definition for ESL is as follows:

> *"The utilization of appropriate abstractions in order to increase comprehension about a system, and to enhance the probability of a successful implementation of functionality in a cost-effective manner."*

This definition does not talk about any specific domain, language, or tool, but instead tries to define the effect that a set of tools and languages will have on the objective of releasing a product that meets all of its functional and non-functional requirements. At the end of the day, managers will adopt something if it improves their productivity and/or quality of results. Another important aspect of the definition is that it implies the transformation between what is intended and what is implemented. It purposely does not include the specification as the ability to create executable specifications is still somewhat in the future.

Industry Taxonomy

What is a taxonomy and why is it useful?

A taxonomy is a way to categorize objects or concepts by showing the relationships that exist between them. It quite often results in a hierarchy or table of attributes, where each of the attributes identifies a particular element of the differentiation.

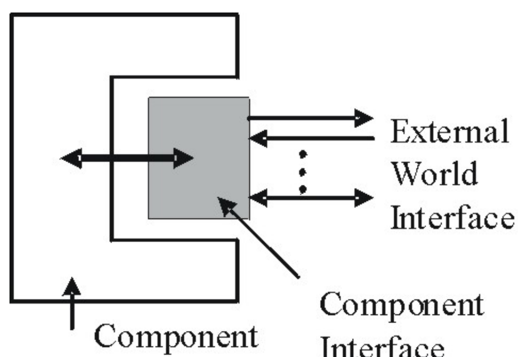
Perhaps the most famous taxonomy is that created by the Swedish scientist Carl Linnaeus who managed to create a classification for all living things. That classification, first published in 1735, is still in use today although with numerous modifications. In 1966 Flynn created a taxonomy for computer architectures that categorizes computers based on their streams of information. A third example is the periodic table of elements, where each column aligns the atoms with the same number of electrons in the outmost orbit that in turn identifies many of the characteristics of the element.

So, why do we need a taxonomy for ESL? Quite simply, because without an identification of all of the elements or attributes of the ESL landscape, it becomes impossible to talk about how certain languages or tools relate to each, and the flows that can be created from them.

Existing taxonomy

In 2005, an EDA industry standard taxonomy[ii] was published in a book that attempted to create precise definitions for many of the industry terms. This book was not the creation of one or two people but from a large number of people coming together and agreeing on those definitions. Part of that book, the model taxonomy, laid out a number of orthogonal axes for defining the attributes of models, namely temporal, structure, data and functional resolution. Some of these definitions define the levels of abstraction and models that are being used in ESL, so a few of the important concepts of this taxonomy will be discussed here. However, in this paper it is not possible to talk about all of the concepts contained within that book.

One important concept in the model taxonomy is that we should talk about the internals of the model and the interface it exposes to the outside independently. It is possible for example, to provide a detailed interface to a model and to define the internal functionality at a much higher level of abstraction. This is shown diagrammatically in Figure 1.



The levels of abstraction available to the external world from a Component Interface may be different to (or a superset of) the level of abstraction of the Component itself.

Figure 1: Internal and External Views

This concept allows the resolution of the model functionality to be modified without having to adjust its interface to the other models that it connects to. Multi-abstraction modeling is a key element of the ESL space.

Attributes axes

The axes explicitly characterize a model's relative resolution of details for important model details. The taxonomy identifies four primary model characteristics:

- Temporal resolution
- Data Value resolution
- Functional resolution
- Structural resolution

While the temporal and data axes are orthogonal to each other, the other two are not completely independent, but they are terms that most model developers feel comfortable with. This is somewhat of a compromise because the ideal situation would be for each axis to be independent of all others. Unfortunately in a world where things evolve, it is rare for there to be a perfect way to cover all cases.

The temporal axis represents the degree of resolution of events on a time scale. Several lables are identified, such as partially ordered events, system accurate, token cycle, instruction accurate, cycle approximate, cycle accurate and gate propagation accurate. While these are arbitrary labels, they serve to identify some of the discrete attributes common to models that exist today.

The data resolution axis deals with the precision with which values are specified. For example, the contents of a register may be represented as an enumerated type, as an integer or as a binary number. The book identifies the following discrete labels: token, property, value, format and bit logical.

The functional resolution axis describes the level of detail with which the model describes the functionality of the model. This could be a mathematical relationship, an algorithmic process or a Boolean/digital logic operation.

Finally the structural axis describes how the model is constructed in terms of its constituent parts, or the hierarchy of the models. A system described in one large block contains no structural details while an RTL description defines the full hierarchy of logic gates and flip flops and how they are connected at all levels of the hierarchy. Clearly the structure of the model can be modified by tools which in general tend to remove or flat layers of hierarchy.

Examples from the model taxonomy

Attempting to look at models in a four dimensional space is clearly not possible, and when adding the complication of the internal and external views a different means of representation has to be found. In the taxonomy book, each of the axes is aligned with each other and the possible ranges or each of these is shown. In general the left hand side is more abstract moving to more detailed on the right. The following convention is used to display the range of abstractions in each of the axes:

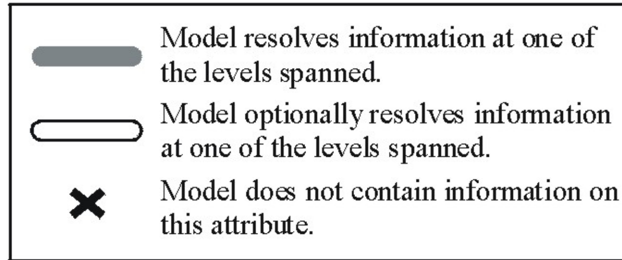


Figure 2: Taxonomy Key

In addition, the taxonomy book also talks about the software programming level, but this will not be discussed here.

- **Functional model**

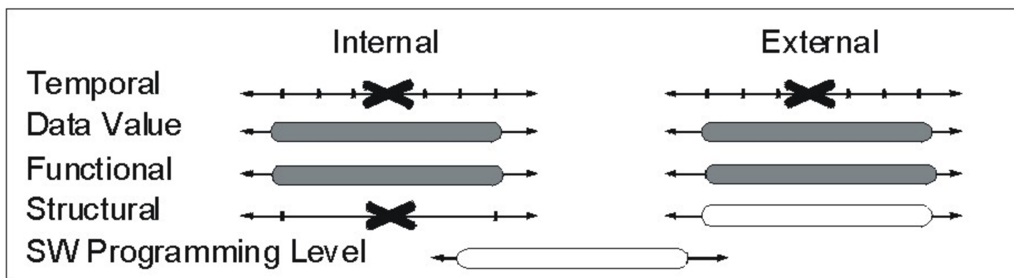


Figure 3: Functional Model

A functional model describes the function of a system or component without describing a specific implementation. A functional model can exist at any level of abstraction, depending on the precision of implementation details. The functional model does not specify any timing other than that implied by dependencies in the function.

- **Behavioral model**

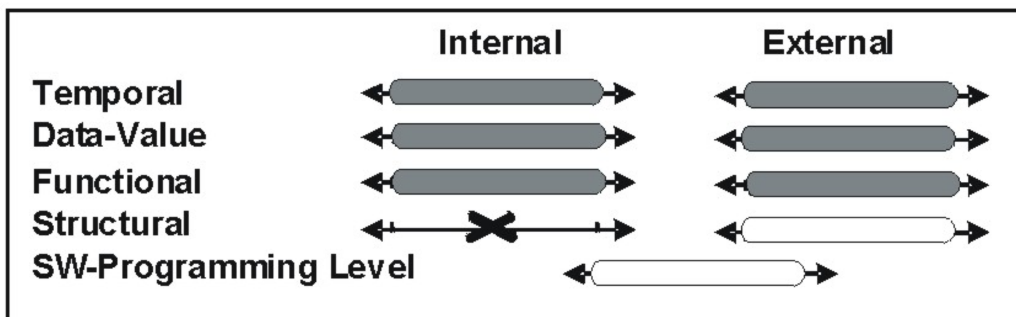


Figure 4: Behavioral Model

A behavioral model describes the function and timing of a component without describing a specific implementation. A behavioral model can exist at any level of abstraction.

We can thus immediately see that while behavioral and functional models are talked

about a lot in the industry with an implied level of abstraction, this is not strictly the case, and the only difference between the two is the existence of timing information.

- **SystemC programmers view with timing**

There are other models that are a lot more specific about the levels of abstraction, such as this one defined in the SystemC TLM. This example is also interesting in that it shows a difference between the internal and externals views of this model. Internally, there are no restrictions placed on how the model is created, except that the definition requires it to be a behavioral model, or in other words have timing information specified at some level. Externally, the model requires fairly close tolerances on how timing and data are specified and requires that it be a structural model that can be instantiated.

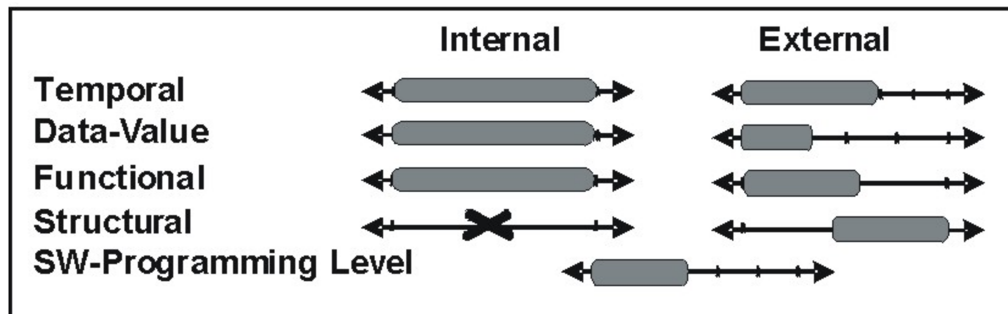


Figure 5: SystemC PVT

Platforms

Platforms have become a key element of system level design for a number of companies. The taxonomy identifies some distinguishing elements between them such as complexity, ranging from a simple collection of blocks, to full systems on a chip. Another distinguishing characteristic is their interfaces, but the biggest differentiator is the way in which they are used. The taxonomy defines three primary use models:

- Technology Driven. This is a bottom up approach where blocks are made available on a particular technology base. There is little pre-integration work done on the blocks, neither are they tuned for any particular application.
- Architecture Driven. This is specified to be a middle out approach. Blocks are pre-integrated and verified and may include an RTOS running on the platform.
- Application Driven. This is the top down approach where decisions are made based on the application itself, and component choices made that directly support the necessary functionality. It generally implies a family of products where a single architecture is created with multiple variants.

Software

A system would not be a system without a processor and that implies an interface into the software world. Software is generally defined in layers rather than the component view of the hardware world. The innermost layer of software is called the hardware dependent software (HdS) layer and includes things such as, the boot layer, drivers, diagnostics, low

level communications and configuration software. Within HdS, four axes are identified for describing such software:

- Life cycle axis. This describes the level of maturity of the software and thus describes its progression through the development cycle. While not something that distinguishes the software itself, it is perhaps something that hardware people should pay more attention to.
- Run-time and real-time axis. This is quite a complex collection of concepts covering both the dynamic state of the software, such as boot, configuration, execution but also when operating, the type of scheduling and scheduling constraints that are being dealt with.
- Hardware architecture axis. This describes the hardware as seen by the software. Its primary function is to take operations within the hardware and to abstract them into discrete functions that can be called from the software. This can exist in many forms including the abstraction of a processor into its instruction set, or even the software used to program programmable logic to perform a certain task.
- Software layering axis. Even within the HdS a number of layers are possible such as the hardware layer, primitive function layer, OS layer and application layer to name a few.

Attributes of ESL

Having looked at the attributes that define many of the components of the ESL space, it is still necessary to identify which of these are important when trying to distinguish ESL tools and methodologies. Each of them-- models, platforms and hardware dependent software -- can have an impact on the overall objective and flow and which pieces of the final solution are created or synthesized from the initial inputs.

The proposed taxonomy is intended to clarify some of these issues by defining the solution space for ESL tools. A taxonomy should be made up of a number of orthogonal axes, and each of these axes should not have any dependence on the others. However, in reality there are engineering and business issues that may mean that not every point in the Taxonomy makes sense, and that there may be some interlinking that goes on between them.

Concurrency

When considering the ESL space, there are a lot of companies trying to provide synthesis solutions, but because the application space and target solutions space are so diverse, it becomes very difficult to compare technologies between different companies or the solutions that they provide. They may also start from descriptions of the design that are at different levels of abstraction, or have different computation models. For example, a design described in the C or C++ languages is a sequential model that describes a solution capable of running on a processor. Some companies have created language extensions that either allow hardware concepts to be included, such as SystemC, or add more fundamental capabilities such as defining possible concurrency, as is done in

HandleC.

Alternatively an algorithmic model may have been developed that makes no such assumption about the underlying implementation and may have concurrency inherently built into the description. An example is a mathematic equation, where dependencies for calculation are directly built into the expression itself. However, while they define dependency they do not explicitly define order.

Given that a software implementation is likely to be a sequential solution (ignoring very coarse grained concurrency defined by threads or some other OS level mechanism) and hardware more concurrent, then it is inappropriate to define which of these models is at a higher level of abstraction. They are just different. A transformation step would be necessary to convert from the algorithmic to the C model and a similarly difficult transformation exists in the opposite direction.

In general synthesis is a process in which resources are created, and operations are scheduled and mapped onto those resources. In that respect it is the creation and management of concurrency. It is clear then that concurrency is a highly central axis in both the input to ESL and the result from it.

Communications

The ESL space is by most definitions a mixture of hardware and software. Most companies in this space today, start with a description that resembles software. This description may be at a number of different levels of abstraction, such as an algorithmic description or as software with the expectation that the software will execute on a processor. In this case the starting point is already one form of implementation of a solution. The tools in this space map all or part of the software into hardware of various kinds, or may leave the application in its entirety on the implied processor, with the tool making that explicit. We can thus see that a wide range of solutions is possible from single processors executing the code, to an all hardware implementation of the solutions, or the solutions mapped into multiple processors etc. Whenever the solution is divided amongst multiple execution engines, there is the possibility for concurrent execution, and this has already been identified as one of the primary axes of the ESL taxonomy. In addition, it is likely that some form of communications will be needed between them, unless they are actually completely independent threads of execution. This communications is the second axis of our taxonomy.

Communications can take many forms and is highly controlled by the architecture of the solution. For example, an all software solution could use shared memory, or pipes which are probably layered on top of shared memory. When software and hardware communicate, it could also be using shared memory, but it could also be tied very closely to the processor as a co-processor. Communications could be through dedicated structures such as FIFOs or the registers between stages of a pipeline. These examples show that while concurrency and communications are orthogonal to each other, there are many places in the solution space that these two axes create that probably do not make a lot of sense. Very fine grained parallelism such as operations within an instruction or dedicated pieces of hardware is naturally handled by point to point communications or pipelines, and not by shared memory. At the other end of the spectrum, two software threads that need to communicate are unlikely to conduct this through resources directly

in the processor.

What complicates matters is that solutions may employ several types of concurrency at the same time. For example a multiprocessor solution may be defined with each processor communicating through a global memory area, but at the same time, each processor may have co-processors which communicate via an internal processor interface and have an instruction pipeline, where buffered communications happens between the stages. Different tools will probably concentrate on particular types or regions of concurrency but there is nothing to say that future tools may not be able to handle a broader range of concurrency types.

Concurrency and communications

While it has already been stated that in an ideal world the axes would be completely orthogonal, there are at times other factors that limits the range of choices. These can be cost, performance or a range of other issues. This can clearly be seen by looking at these two axes together.

Figure 6 shows some of the more common points on these axes where solutions exist. As can be seen not all points are filled in as the coarser levels of concurrency can afford to use more generic, shared forms of communications, whereas very fine levels of concurrency need more dedicated, faster access methods.

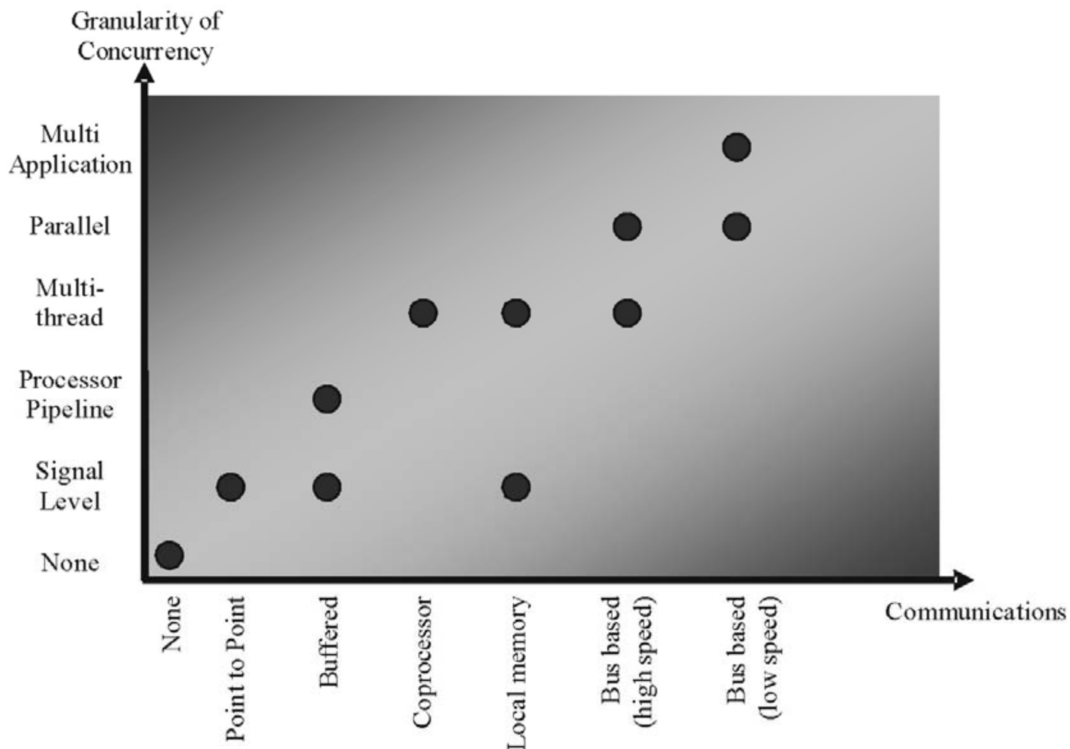


Figure 6. Common Solutions Showing Concurrency and Communications

Configurability

However, the story does not end there as there is another important axis necessary to

define ESL solutions and that is the configurability of the solution. Software can be considered infinitely configurable, but it is the configurability of the platform that the software runs on, or is embedded in, that is being talked about here.

Some solutions such as a dedicated hardware solution are very rigid and can only perform one task. A processor with a fixed instruction set would also be considered a fixed solution. However a processor with an extensible instruction set has some degree of configurability. This configurability is handled by the tools in the design path. Hardware can also be built to be configurable. For example a SerDes I/O block can be configured to support any number of different serial protocols and many communications processors can also support the higher levels of the protocol for these communications mechanisms. Finally, a reprogrammable fabric such as an FPGA or PLD can be reprogrammed at startup, or even possibly reprogrammed during operation of the system. These categories are, in very coarse terms: fixed, configurable at design, configurable at startup or dynamically configurable.

This axis is orthogonal to both of the other two axes, although commercial solutions available do bind solutions to some of the other axes. For example an FPGA only has available to it certain means of communications (signal or memory based, unless it has an integrated processor).

Some Examples

In this section some examples will be given for common models and for places in the matrix that would be familiar to designers today. Then it will address a few example commercial tools in respect to the transformations they make from the initial input description and what they create. Similar to the original model taxonomy, the three axes will be shown in a horizontal chart with the most abstract being at the left and more refined to the right.

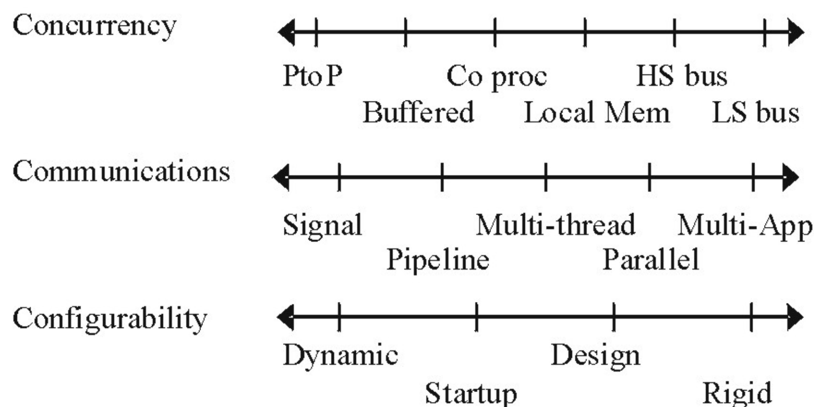


Figure 7: ESL Attribute Axes

Point solutions

It should be remembered that these ESL attributes do not imply anything about the abstraction of the model. They can still be represented as any of the model types defined within the model taxonomy so long as the abstraction has the ability to describe the

capabilities represented here.

- **Hardware languages**

A pure hardware solution written in Verilog or VHDL is most likely to use point to point or buffered communications with tight communications between each of the elements. It is possible to define complete systems that would contain multiple processors using these languages, but in this capacity they are being used as structural interconnect for building blocks probably defined at higher levels of abstraction. The intent here is the type of input that could be fed into a synthesis product to produce hardware. The complete range of configurability is possible since programmable logic such as an FPGA can be created, or it can be defined for a single application.

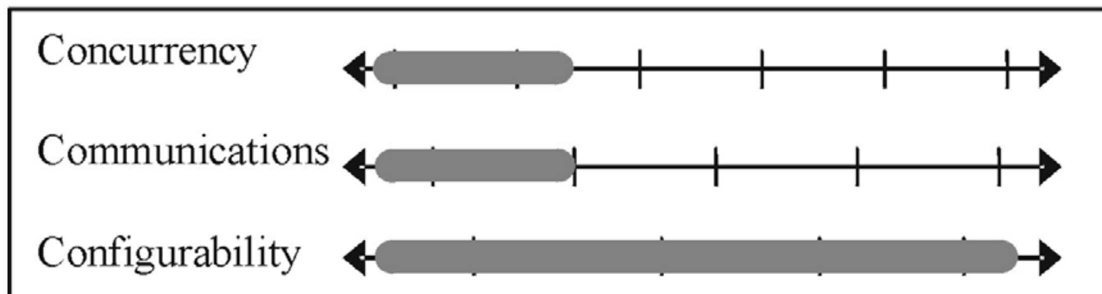


Figure 8: HDL description

If we were to consider an FPGA itself, then the configurability is performed either at startup or in some cases can be done dynamically. Internally to the FPGA, communications may either be via signals, although this is limited due to timing issues, via buffers or through local memory. Concurrency is still predominantly fine grained. This is shown below.

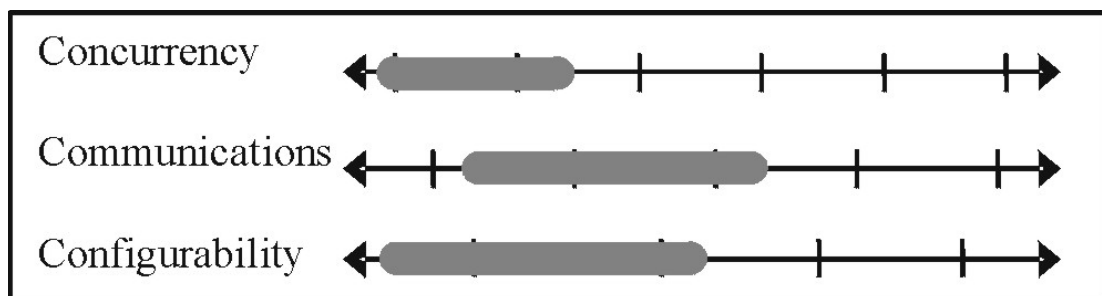


Figure 9: FPGA Solution

- **Software written in C**

The C language has no built in constructs for dealing with concurrency or communications as these are capabilities provided by an operating system. Thus when considering C as an entry vehicle, it is a highly restricted language which is why many variants of it have been created to deal with concurrency in particular.

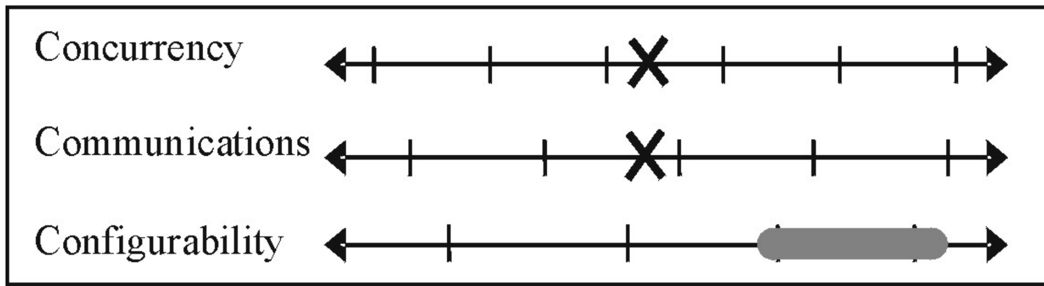


Figure 10: Generic C

As an example, HandleC, a C variant developed at Oxford University in England directly adds concurrency and inherently provides memory communications as variables can be shared across the parallel execution threads that are created. HandleC is shown below.

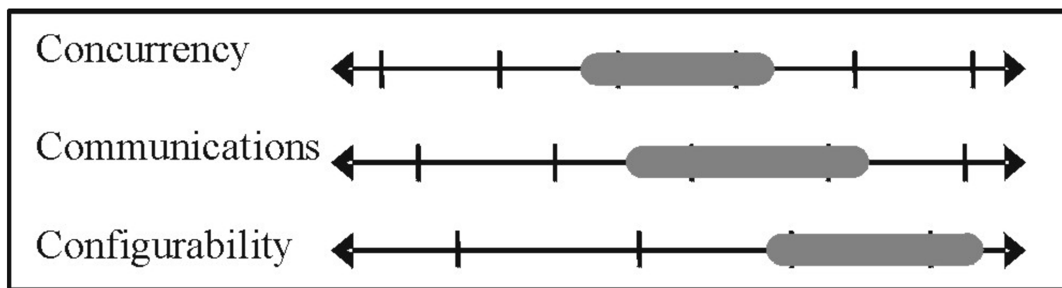


Figure 11: HandleC

Another popular C derivative is SystemC. This language also adds concurrency, although in a very different manner than HandleC, and also adds several explicit means of communications ranging from signal based, and messaging through to shared variables, although this is perhaps not a recommended method as the language does not provide safe access methods to those variables. SystemC is shown below.

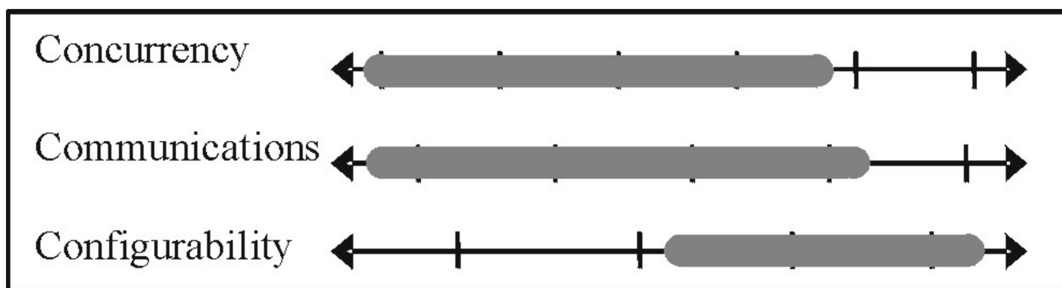


Figure 12: SystemC

Popular solutions

Many of the tools on the market today tend to map to a few distinct architectures such as a processor plus some logic, where the logic is automatically created to perform a part of the task that was found to consume a large percentage of the time. Most of these applications are algorithms where there are a few tight loops at their core. However, there are a number of ways in which this logic can be attached to the processor. It can be a very close coupling such that the added logic becomes an instruction in the processor itself, directly sharing resources with other instructions. These are the extensible processors

whose attributes are shown below.

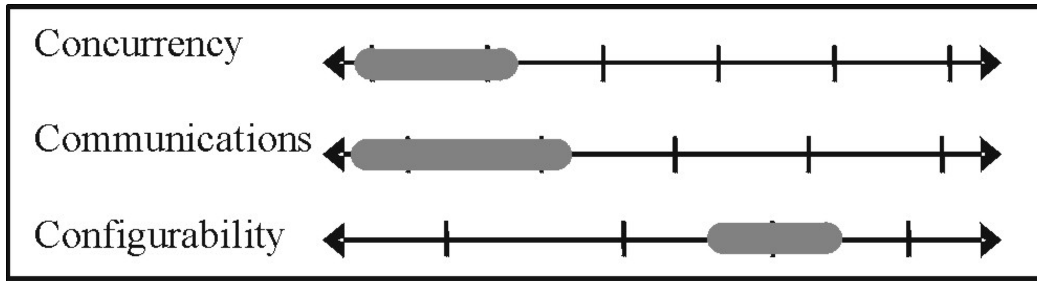


Figure 13: Extensible Processor

More common are the solutions where a co-processor is created. Most of these solutions today are not good at creating concurrency, but are looking at speeding up part of the problem such that it has a significant impact on the total execution time. Alternatively, by reducing total execution time, it may become possible to slow down the main processor such that the gain is translated into lower power. This class of solution is shown below.

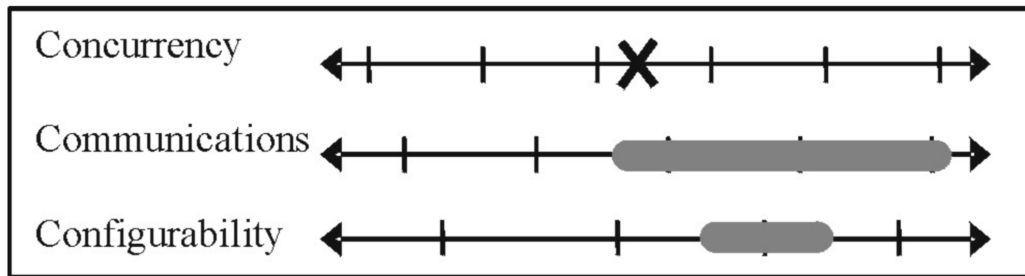


Figure 14: Processor Plus Co-Processor

Over time, it is likely that these solutions will add some degree of concurrency. It is also possibly that co-processors will be generated that are capable of more than one function given them greater configurability.

Tools and Flows

Given the ranges of points defined above, the reader should now be able to see the magnitude of the differences between certain types of ESL tools. Depending on the starting point, and the ending point, they may be adding concurrency, or defining communications and the platforms that they target will have very different capabilities. At the same time, certain tasks or goals are specific enough that very focused solutions can be defined for them, and while they are targeted at smaller audiences, they have the ability to provide real value to those people. Shown below are just two sample tools. These two have been chosen to show the extreme differences, and do not imply anything about those solutions.

- **Poseidon Design**

This company starts from a generic C description and performs analysis on the code, identifying the parts of it that can be migrated from a general purpose processor onto a highly targeted co-processor. The solution has the ability to target different levels of communications between the processors.

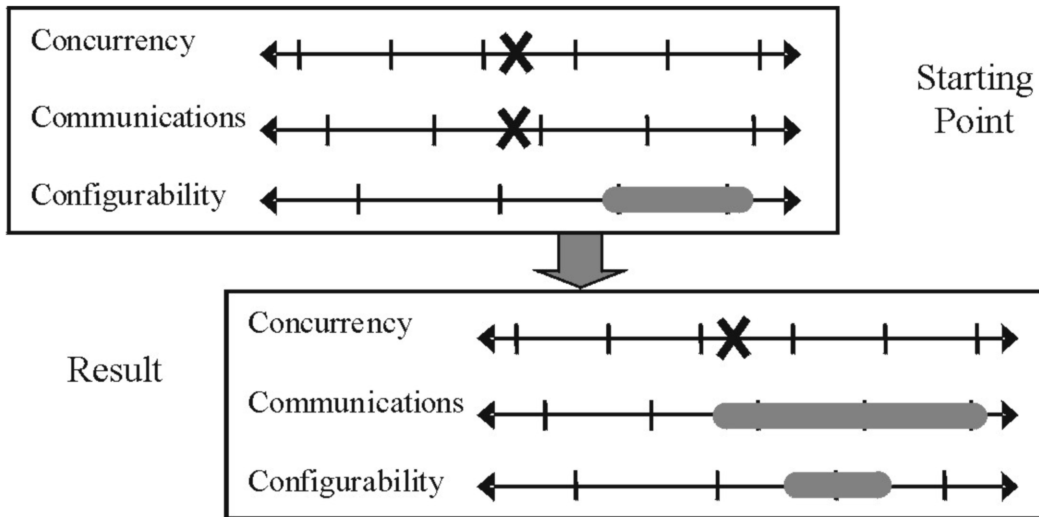


Figure 15: Poseidon Design Transformation

The co-processor that is created will then need to be synthesized to create the final design.

- **Forte**

Almost everything about Forte is different from Poseidon Design. They start from a SystemC description and perform what could be described as more classic synthesis with the aim of producing hardware that can implement the desired functionality. Their transformation is shown below:

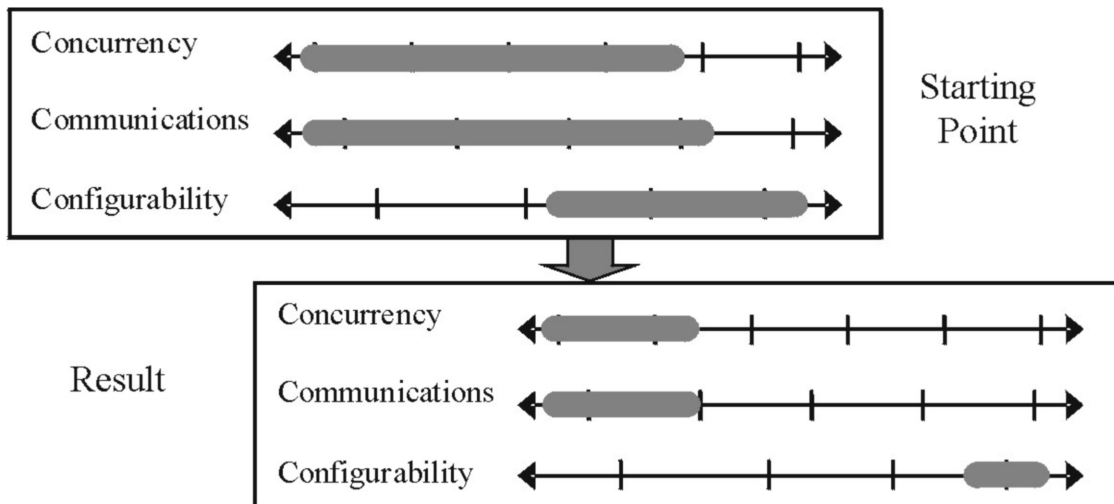


Figure 16: Forte Transformation

Conclusions

In this paper, a taxonomy has been defined for ESL based on three primary axes: concurrency, communications and configurability. By associating solutions and tools into these axes it becomes possible to find the overlap and similarities between tools and also to serve as a guide for which tools will work together to produce complete flows. While this paper has not presented all of the possible flows, tools or ways in which the models can be used, it has provide the reader with the information necessary to perform that task themselves and to be able to compare the tools that exist in the market today. I actively encourage discussion on this subject so that the industry can come to agreement on these, or alternative axes, for defining this emerging field.

References

---

[i] Gary Smith, Daya Nadamuni: "ESL Landscape, 2005" Gartner research, May 2005

[ii] Bailey, Martin and Anderson: *Taxonomies for the Development and Verification of digital systems,* Springer, 2005