Meta-Data-Enabled Reuse of Dataflow Intellectual Property for FPGAs

Adam Arnesen

NSF Center for High-Performance Reconfigurable Computing (CHREC)

Dept. of Electrical and Computer Engineering

Brigham Young University

Provo, UT, 84602, USA

adamarnesen@byu.net

Abstract—This paper demonstrates the ability to reuse arbitrary IP as primitive cores in architectural synthesis algorithms for FPGA by encapsulating these IP in meta-data. This meta-data is represented as a set of extensions to the IP-XACT XML specification and defines the high-level data types and the temporal behavior of IP. This paper describes how these extensions are used in the Ogre synthesis system to facilitate automatic synthesis of control and interface logic for homogeneous synchronous dataflow (H-SDF) designs.

I. Introduction: High-Level Synthesis

High level synthesis (HLS) is the process of automatically creating digital circuits by from an abstract behavioral specification of a digital system and finding a register-transfer level (RTL) structure that realizes the behavioral specification [1]. These behavioral specifications can be done in a traditional software language such as C and then translated into a high performance hardware system [2], [3].

This work describes a method for using coarse grain intellectual property (IP) as primitives in HLS. HLS algorithms generally map behavioral specifications to small-sized low-level primitives such as adders and multipliers. When more coarse-grain IP have been used for synthesis, the set of possible IP was often limited to a small set of IP that was native to the synthesis tool. This work demonstrates a technique that allows any coarse-grain IP to be used in synthesis thus allowing the set of operations for synthesis to include any arbitrary IP from any source.

Arbitrary coarse grain IP can be used in HLS if additional information is provided to the HLS tool about the IP. This work encapsulates this information, or meta-data, by using the IP-XACT standard and extensions describing the high-level numerical datatypes and temporal behavior of the IP [4], [5], [6]. High-level datatype information enables tools to assist a designer in ensuring that datatypes are correctly manipulated as data is transmitted between IP. Meta-data specifying the temporal behavior of IP allows tools to automatically synthesize control circuitry to enforce the proper sequences of IP operation and ensures that no data is lost in communication.

This paper will discuss a design tool known as Ogre that used datatype and temporal behavior specifications in meta-data to enable the synthesis of dataflow systems. Ogre utilized the Simulink GUI and model file to represent models of systems composed of reusable IP as shown in Figure 1. These models were translated into FPGA designs using the Ogre design flow shown in Figure 2. This flow could reason about high-level datatypes and temporal behavior and synthesize control circuitry. The use of high-level datatypes in Ogre will be discussed in Section II. Section III will discuss how Ogre used meta-data to represent coarse-grain IP as actors in H-SDF. Section IV will discuss how synthesis techniques were applied to these coarse grain IP to create complete systems and Section V will conclude.
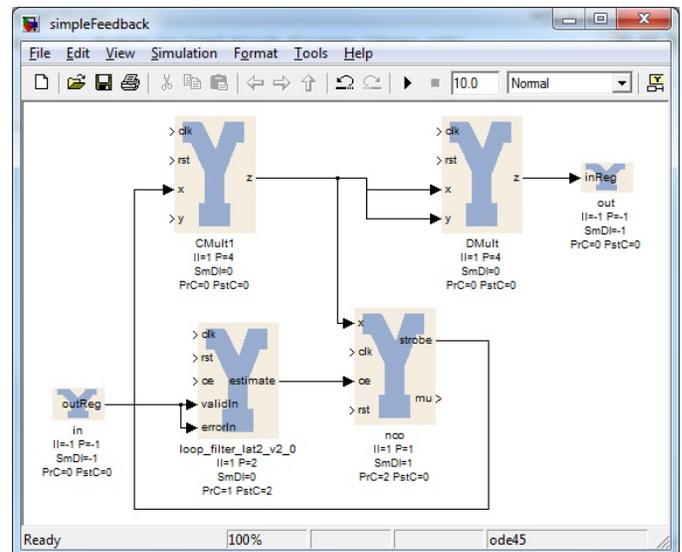

Fig. 1. Simulink is used as a front end for design entry by the Ogre tool. IP blocks described in IP-XACT XML are automatically included in a Simulink library and can be dropped onto the simulink design pallet to create complete designs.

II. Numerical Datatypes

The Ogre tool used high-level numerical datatypes to check for valid data transfer between IP that have been connected in
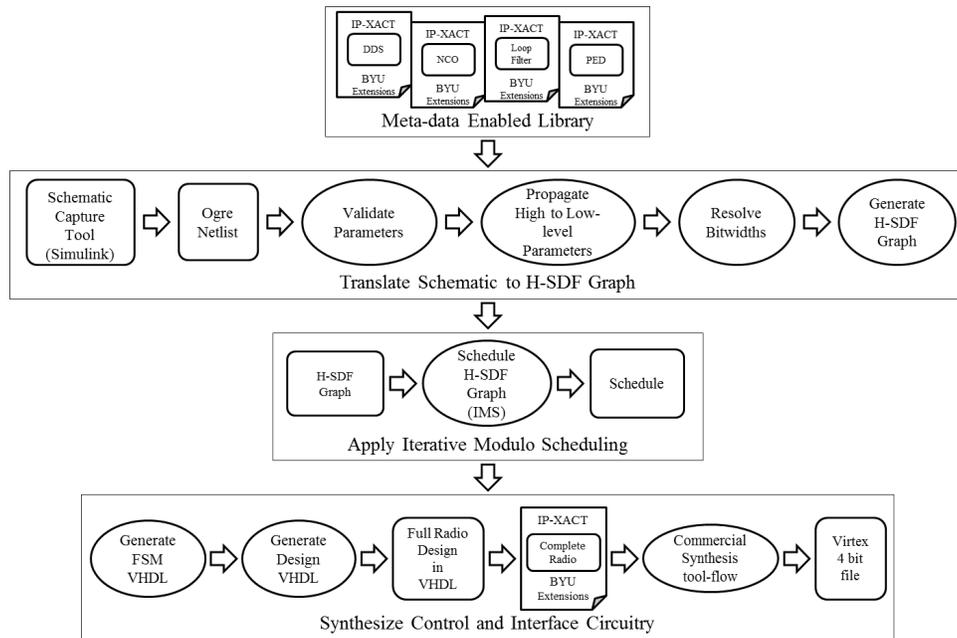
Fig. 2. An overview of the Ogre System. There are four primary components: library representation, translation of schematic information to H-SDF graphs, scheduling the H-SDF graph, and synthesizing control and interface circuitry to create a complete downloadable bitstream.

a design. Much of the IP that is used in data-flow designs for FPGA operates on numerical data. Because of this, signals in data-flow computations are also often meant to be interpreted as some type of number such as an integer or a fractional number represented as fixed or floating point. When composing data-flow designs and attempting to reuse IP, it is vital to know the mapping of bits in a signal to the numerical data being represented by this signal. If this mapping data is not available, blindly tying cores together will almost certainly result in incorrect data transmission. Ogre utilized the meta-data descriptions developed as extensions to IP-XACT which define a standard way of mapping high-level datatypes to their underlying bit-vector implementations.

A. Representing Datatypes

The extension to IP-XACT that allows for the representation of datatypes defines mappings between high-level datatypes and the bits that implement those types on a particular signal. There are two components to this representation: the definition of the underlying bit-vector and the mapping of these bits to a high-level type.

The bit-vector representation is contained in the meta-data description of each port in XML. The port is defined by the XML ``vector'' element with the width of the port being defined as the left side of the vector minus the right side of the vector ($left - right$).

The high-level type is defined separately from the description of the port. Each port points to the high-level type that should be used to represent it allowing multiple ports to be defined by the same type without the need to duplicate the description of the type. An example of a high-level type

XML Code 1 This code snippet shows an example of a high-level datatype extension. This example shows a signed fixed-point type where two bits are used to represent the integer part.

```
<spirit:component>
  . . .
  <spirit:vendorExtensions>
    <chrec:highLevelDataTypes>
      <chrec:portDataType>
        <chrec:name>SFix_2_a</chrec:name>
        <chrec:fixedPoint
            chrec:sign="2sComplement">
          <chrec:intBits chrec:resolve="static">
            2
          </chrec:intBits>
        </chrec:fixedPoint>
      </chrec:portDataType>
    </chrec:highLevelDataTypes>
    . . .
  </spirit:vendorExtensions>
  . . .
</spirit:component>
```

definition is shown in XML 1. This particular definition is for a fixed-point datatype. It defines that the two most significant bits should be interpreted as integer bits and the rest of the bits of a signal should be interpreted as fractional bits. For fixed-point datatypes it is also possible to specify the number of fractional bits that should exist on a signal and assume that the rest are integer bits. The extensions to IP-XACT for datatypes and the details of their implementation are discussed at length in [6] and [5].

B. Utilizing Numerical Types

The IP-XACT extensions developed in this research that describe high-level types were used in the Ogre design synthesis system to assist designers in ensuring that datatypes matched between IP. Ogre ensures first that bitwidths between IP match and then checks the datatypes of these connections. If there is a mismatch, Ogre alerts the user to the problem.
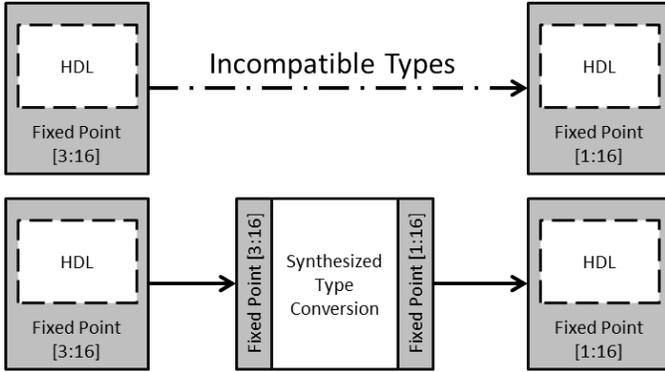

Fig. 3. Datatype-aware tools can use high-level datatype information to synthesize conversion logic between incompatible datatypes and thereby ensure correct data transmission.

Although the functionality of Ogre leveraged high-level types only for checking of proper data connections, these meta-data defined types provide the ability for a tool to perform more sophisticated datatype synthesis. For example, a tool could leverage the datatypes in meta-data to automatically synthesize datatype conversions between IP when it detects that there is a datatype mismatch. When a tool detects a mismatch, a parameterized block of IP could be inserted between the incompatible ports, making their datatypes compatible as shown in Figure 3.

The high-level datatypes developed in this work are essential for any tool that will automatically compose cores for dataflow designs on FPGAs. If the high-level numerical datatypes are not defined, there will most likely be corruption of data as it moves between IP in a design. The meta-data high-level types included in CHREC XML and in extensions to IP-XACT provide the necessary mapping between the bits of an IP port and the high-level type which that port's data belongs to.

III. Representing Coarse-Grained IP as H-SDF Actors

In order to automatically compose arbitrary IP in a dataflow system, a description of the timing behavior of the IP's interface is required. If temporal core behavior for IP can be matched to a particular model of computation, tools will be able to reason with these cores and automatically generate control circuitry for designs. The meta-data proposed in this work to describe timing behavior is based on the homogeneous synchronous dataflow (H-SDF) model of computation [7]. This section will briefly describe the H-SDF model and describe how meta-data implemented as extensions to IP-XACT enabled cores to be mapped as H-SDF actors.

A. The H-SDF Model of Computation

The H-SDF model of computation defines the execution semantics for a system based on the dataflow relationships between portions of the system. H-SDF is represented by a directed, vertex weighted, graph $G = \{V, E\}$. Each vertex $v \in V$ is called an H-SDF actor and each edge $(x, y) \in E$ represents the operation precedence between two actors.

The edges in E are used to enforce execution semantics on the H-SDF graph. For example, the presence of edge (a, b) in G means that all computation must be done in vertex a before vertex b can start its computation. Computation in H-SDF is done by the actors. When an an actor performs a computation, it is said that the actor "fires." The weight of the vertex v represents the number of steps required for that particular actor to fire.

In addition to simply defining edges between vertices, H-SDF also uses the notion of tokens to enforce semantics. Each edge in G can contain multiple tokens at any given time. In order for any actor in H-SDF to "fire," or perform computations, it must have an input token on each of its inputs. If there are no input edges to an actor, it may fire at any time. When an actor "fires" it produces tokens on all of its output edges.

Homogeneous synchronous dataflow is a subset of standard synchronous dataflow (SDF) because when H-SDF actors "fire" they consume only *one* token from their inputs and produce *one* token on their outputs. In general SDF, actors are allowed to consume and produce multiple tokens when they fire(i.e., multi-rate dataflow). In this work H-SDF was chosen as the model of computation because many single-rate systems can be represented as actors that consume and produce single tokens when they perform computations. H-SDF was also chosen because it is easier to use than general SDF.

Figure 4 shows an example of an H-SDF model and a valid sequence of actors firing. Because actors A and B have no inputs they can fire at any time. When they fire, they each produce tokens on their output edges. Actor C is allowed to fire once tokens produced by A and B are both present on its inputs. When C fires, it also produces a single output token which is consumed by actor D when it fires. Actor D's firing completes a valid computation from this H-SDF model.

The H-SDF model of computation also supports cyclic data dependency graphs. However, when an H-SDF graph is cyclic, care must be taken to correctly satisfy the initial conditions for the computation. For each cycle in an H-SDF graph there must be at least one token on an edge in that cycle. If there is no token in the cycle, the computation will not be able to start because no actor will have the needed inputs. Multiple tokens may exist in the cycle or even on a single edge, but at least one token must be in the cycle. An example of proper and improper initialization of H-SDF graphs is shown in Figure 5.

The execution semantics of H-SDF allow a static schedule to be computed for the graph. This schedule will be a repeating schedule that defines the relative start times for each of the
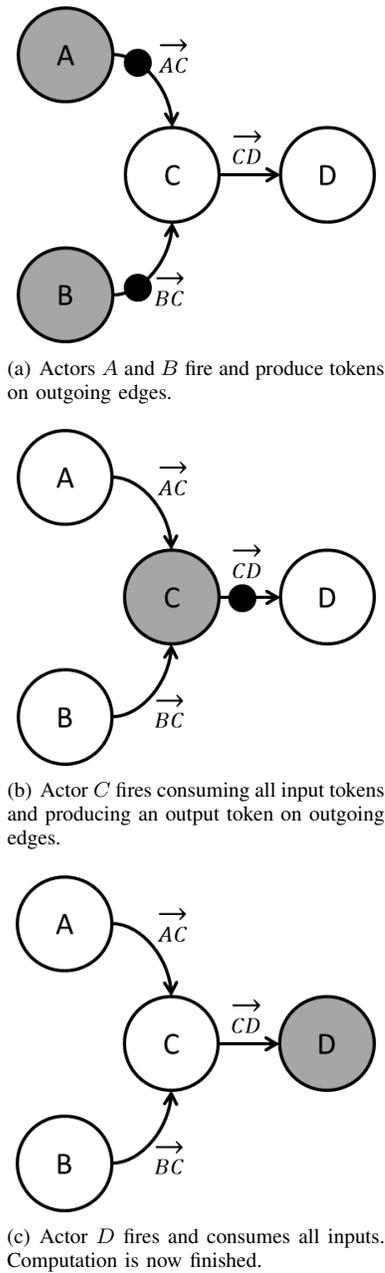
(a) Actors *A* and *B* fire and produce tokens on outgoing edges.

(b) Actor *C* fires consuming all input tokens and producing an output token on outgoing edges.

(c) Actor *D* fires and consumes all inputs. Computation is now finished.

Fig. 4. The homogeneous synchronous dataflow model of computation allows each node to "fire" when one token is available on each of its inputs. Each firing produces one token on the node's output.

(a) Invalid H-SDF initial conditions

(b) Valid H-SDF initial condition with one initial token in the loop

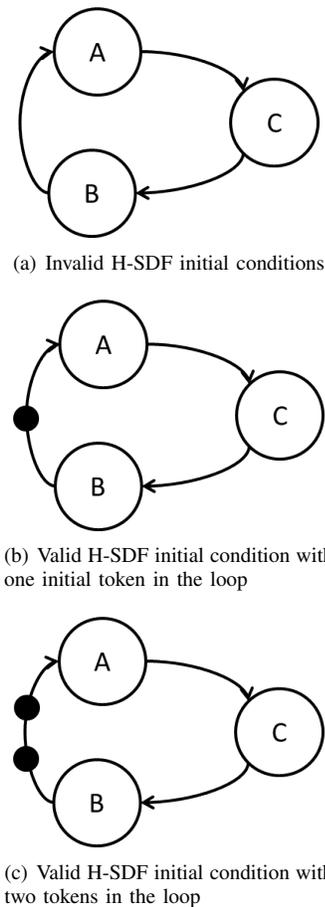(c) Valid H-SDF initial condition with two tokens in the loop

Fig. 5. A cyclic H-SDF graph must have proper initial conditions. Each cycle in the graph must start with at least one token already on an edge in the cycle. The graph in figure 5(a) is invalid because it has no such initial condition. Figure 5(b) shows the simple case of a valid initial condition with one token initially in the loop. Multiple initial tokens are valid as shown in Figure 5(c).

actors. When using H-SDF to represent hardware systems, the schedule can be mapped on to clock cycles for pipelined IP.

Because H-SDF enforces execution semantics on a dataflow graph, it is useful in describing the execution of single-rate dataflow systems for FPGAs. If each IP core in a system can be interpreted as an actor using H-SDF semantics, then the execution semantics of H-SDF can be used to determine how the hardware system should execute by creating a static schedule for the operation of IP in the system. This research defines three meta-data elements that allow coarse-grain IP

to be described in a way that allows them to be interpreted as actors onto the H-SDF graph. This meta-data defines the latency, the data introduction interval, and the sample delay for IP.

B. Latency

The latency of an IP core is the number of clock cycles that elapse from the time that data is consumed on the inputs of the core to the time that the corresponding results are produced on the outputs. This does not mean that the core is pipelined in the traditional sense or that data can be accepted by the core on every cycle. For example, cores that accept data only every 8 cycles and take 9 cycles to compute a result would be given a latency value of 9.

When mapping a IP core onto an H-SDF actor the latency of IP is represented in H-SDF graph by the weight of an actor. Because this weight defines the amount of time that elapses while an actor is performing a computation, this weight can interpreted as the number of latency clock cycles. This information can be used by H-SDF scheduling algorithms to to determine the time that data will appear on the output of

a core. The latency can also allow synthesis algorithms to appropriately control IP downstream to wait until valid data has been produced by the IP.

C. Data Introduction Interval

The data introduction interval for a core describes how many clock cycles must elapse between the introduction of data for each new sample. Cores with a data introduction interval of one can accept new samples each clock cycle. The data introduction interval of a core is independent of its latency. For example a core that has a data introduction interval of 3 can consume data on clock cycle 0 but then will not consume data again until clock cycle 3 and then again on cycle 6. This same core may take 9 cycles to compute a result from a set of inputs.

The data introduction interval imposes an additional constraint on the scheduling algorithms that generate control for H-SDF execution. If tools are aware that a core can only accept new data every n clock cycles, then any synthesized control circuitry must ensure that data is given to a core only when it is able to receive it.

D. Sample Delay

Sample delay is perhaps the most complected parameter used to describe IP as H-SDF actors. The sample delay is the number of cycle iterations separating an actor from the downstream actors. In other words, the sample delay defines how many cycle iterations later the data produced by an IP will be needed for computation. Sample delay is important when IP are going to be used in a cyclic manner. For example, in the design shown in Figure 1 there is a cycle in the design. The sample delay defines the break between iterations of the cycle.

Sample delay can also be thought of as the number of initial tokens in a cycle in an H-SDF graph. If we consider a design to be represented as a H-SDF graph, we know that there must be at least one initial token on an edge in the cycle in order to enable this loop to execute properly according to H-SDF semantics. It is this initial condition that the sample delay represents. The sample delay parameter indicates the number of H-SDF initial tokens existing on the outputs of a particular IP.

Another way to conceptualize sample delay is that it represents the state generated by the previous iteration of the loop. For example in Figure 5(b) the token that exists on the arc $\vec{ba}$ represents the result of the computation done by the previous execution sequence $\{a, c, b\}$.

This research chose to represent sample delay as a property of a piece of IP. While the proper way of representing sample delay in cyclic models is an open research question, there are several advantages to representing it as a property of a particular IP block. Representing sample delay as a property of a block of IP is especially useful when IP cores are used in situations that are closely related to their original design. When the sample delay is a property of particular communications IP, for example, these blocks need only be inserted in a cycle and

the sample delay of that cycle is automatically satisfied. This type of representation is also logical when thinking of sample delay simply as the state from the previous iteration of the cycle. If the IP with the sample delay also has internal registers to maintain state, these registers contain the result of all upstream computation in the cycle. This type of representation, however, is now without its weaknesses. It breaks down if a core is used in a situation that is not similar to its original use. This may cause the sample delay on the IP to be in an incorrect location.

E. IP-XACT Extensions for H-SDF

The meta-data description elements needed to represent coarse-grain IP as actors in H-SDF was implemented as a set of XML elements called the ``behavioral layer''. This set of elements was added to IP-XACT as a vendor extension. XML 2 shows the definition of a temporal H-SDF interface as it appears as an IP-XACT extension. This particular interface has a data introduction interval of 7, a pipeline depth of 8, and a sample delay of 0. This representation method allows sample delay to be represented as part of the IP core and does not require the user to understand the complex concept of sample delay.

The IP-XACT extensions representing H-SDF interfaces enabled scheduling and synthesis algorithms to be applied to IP that had this type of interface. These types of algorithms allowed hardware to be automatically synthesized to control the flow of data between the H-SDF cores.

IV. Applying H-SDF Synthesis Techniques to Coarse-Grain IP

The description of coarse grain IP as actors in the homogeneous synchronous dataflow model of computation allows traditional architectural synthesis algorithms to be used to synthesize control logic for systems. Although the algorithms used for this synthesis have been used before, the meta-data presented in this work enables coarse-grain IP to be used as primitives in these algorithms.

There are several assumptions made in Ogre about the structural interfaces of the IP that will be used to create designs. All IP must have a fixed latency and all inputs must be consumed on the same clock cycle. The latency may be parameterized, however, once a particular instance of the IP exists the latency must be the same for every computation done by that core. Ogre also assumes that each IP has two control signals: `clock-enable` and `data-valid`. These signals are used by synthesized control circuitry to properly start and stop IP operation.

This section will describe the method used in the Ogre tool to leverage meta-data to perform architectural synthesis. The Ogre tool leverages the Mathworks' Simulink tool as an input method for data-flow designs as shown in Figure 1. Once IP have been connected in the Simulink GUI, the Ogre tool can parse the .mdl file and use the meta-data describing the IP to perform synthesis of complete designs. An overview of this synthesis flow is shown in Figure 2. Details of the

```
<chrec:behavioralLayer>
  <chrec:dataIntroductionInterval>
      7
  </chrec:dataIntroductionInterval>
  <chrec:pipelineDepth>8</chrec:pipelineDepth>
  <chrec:sampleDelay>1</chrec:sampleDelay>
</chrec:behavioralLayer>
```

synthesis flow will be described in this section. The method of constructing an H-SDF dataflow dependency graph will be presented as well as an overview of the iterative modulo scheduling algorithm that was used as a first step toward creating control circuitry. The method of converting schedules to finite state machines will also be presented.

### A. Translating Schematics to H-SDF Graphs

Before architectural synthesis algorithms could be applied to systems composed of IP described in meta-data, the interconnection of the IP were represented as an H-SDF graph. This translation used the structural interconnection between IP described in the Simulink GUI and the meta-data describing each of the blocks to create the H-SDF graph.

An intermediate netlist data structure was used as part of the translation that represented the connectivity of the complete design. This netlist structure, shown in Figure 2 as the Ogre Netlist, represented all of the data that was contained in the extended IP-XACT meta-data. The netlist structure was connected to a library of IP meta-data that it queried to determine the structural interface of a core, its parameters, its datatypes, and its temporal behavior. Each of these description elements was encapsulated in an "instance" of each core in the design. Each of these instances contained ports that could be connected in the netlist structure to represent a full or partial design.

The first step in translating a Simulink model file to an H-SDF graph was to populate the Ogre netlist structure with instances of the IP that were in the design and to connect the data ports as represented in the model. Many of these IP were parameterized, and the parameter values set in Simulink were translated into the IP instance in the Ogre netlist. Once parameter values were set, Ogre verified the correctness of the provided parameter set by using the mathematical expressions provided in the meta-data. Once these parameters were validated, mathematical expressions were used to properly set all of the low-level parameters on each of the IP instances.

Datatype checking was a two-step process. First the bitwidths were set to be compatible across the design. The bitwidths set by the user on the input ports in Simulink were used as a starting point to propagate the bitwidths throughout the design. Many of the IP used had parameterizable bitwidths. Because of this parameterization, making bitwidths compatible was often a simple matter of setting the correct parameter to match bitwidths. When parameters could not be set to correctly resolve bitwidths, Ogre would report this to the user who would have to resolve the conflict. While resolving

bitwidth values in the design, Ogre also checked for conflicting high-level datatypes. Mismatches identified using the datatypes defined in meta-data were also reported to the user.
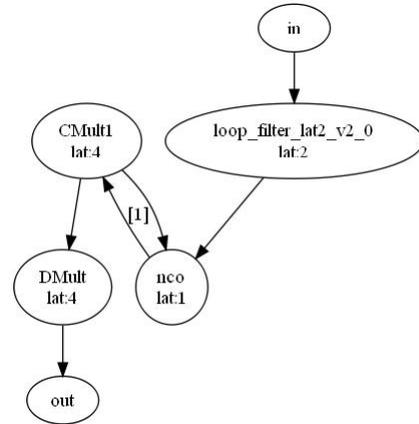
(a) The initial translation of design to H-SDF represents initial conditions (sample delay) as a distance on the edge after the IP with the sample delay. Node weights reflect the latency of IP.
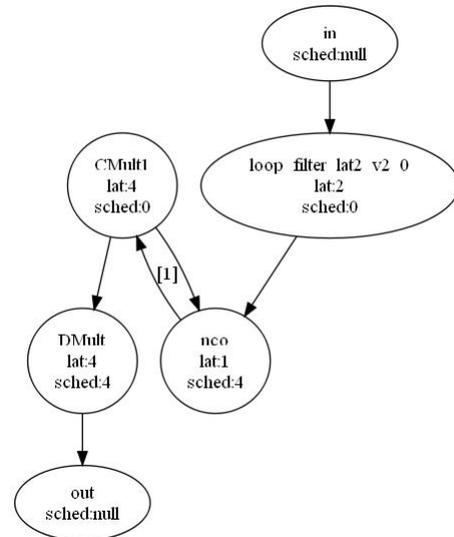
(b) After scheduling, nodes are annotated with the start time determined by the iterative modulo scheduling algorithm.

Fig. 6. The 2 H-SDF graphs shown here represent the graph that is created to represent the design shown in Figure 1. Before scheduling, only weights and sample delay are present on nodes. Scheduling applies a start time to each node.

Once the Ogre netlist was completely populated from the

Simulink description, an H-SDF graph was produced that represented the temporal behavior of the computation system defined in Simulink. For each of the instances of IP in the netlist an H-SDF actor was created. The weight of this actor was the latency of that particular IP. For each wire in the netlist the corresponding edge was created in the H-SDF graph. These edges were weighted according to the sample delay description of their source actor. Because sample delay occurs infrequently in blocks, the weight of most edges was 0. Input and output ports were also represented as actors but their weight was always 0. They were include only as a means for determining consistent starting position for the scheduling algorithm discussed in subsection IV-B.

An example of the result of this translation is shown in Figure 6. This particular example shows the H-SDF graph that results when the design shown in Figure 1 was translated to H-SDF. Note the translation of the pipeline depth to the latency or weight of each of the nodes. Also note that because the "nco" has a sample delay value of 1 there is a weight of 1 applied to the edge from the "nco" to "CMult1."

The translation from dataflow block diagram to H-SDF graph was enabled by the Ogre netlist structure that was based on meta-data contained in the extended IP-XACT specification developed in this work. This meta-data allowed a simple H-SDF graph to be created that represented coarse-grain IP from a library to be represented as actors in H-SDF and allowed a correct representation of the connections between them in the data path of a design.

B. Applying Iterative Modulo Scheduling

The H-SDF model of computation defines its execution semantics, but the implementation of these semantics must be properly represented in hardware in order to produce a working design. As a first step in translating the semantics of H-SDF to hardware, Ogre applied a scheduling technique to determine the relative start times of each of the IP in a design based on a global clock signal. Ogre used an iterative modulo scheduling (IMS) approach to this schedule as described in [8].

The IMS algorithm described in [8] was intended for general scheduling of multi-cycle actors onto available processors. Ogre simply needed to determine the clock cycle that data would be ready for each IP to use to do computation. The power of IMS for this use was that IMS computed a minimum initiation interval (II) for cyclic H-SDF graphs. The initiation interval was the number of clock cycles that must elapse between times that the design was able to consume new data. Minimizing the initiation interval was important because lower initiation intervals corresponded to higher throughput for cyclic data-flow designs. Because many of the designs developed for this work were cyclic in nature, this was an appropriate algorithm choice.

Many scheduling algorithms, when applied to cyclic H-SFD graphs, did not allow for the minimum initiation interval. For example, many algorithms required that a complete computation through the graph be complete before beginning a new computation as shown in Figure 7(a). IMS allowed a schedule
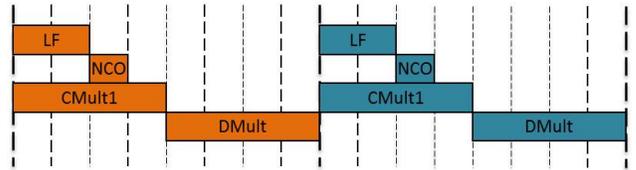
(a) Most common scheduling algorithms require that a completed computation iteration be finished before beginning another. The II produced by this type of scheduling generally does not produce the minimum II, in this case the II=8
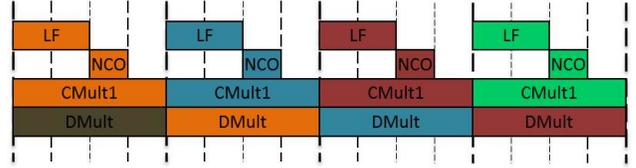
(b) The IMS schedule allows for iterations of a loop to overlap each other. Each color represents the progression of a complete computation through the H-SDF Graph. This produces a schedule with a II=4 which is much better than the minimum possible for the H-SDF graph shown in 6(a)
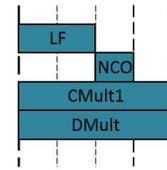
(c) The kernel of the IMS schedule represents the repeated start times for IP in a cycle.

Fig. 7. Scheduling possibilities for the design shown in Figure 1 and the H-SDF graph shown in Figures 6(a). Scheduling algorithms determine the start times for H-SDF actors. The iterative modulo scheduling algorithm defines the start times for IP in a kernel that allows iterations of a cycle to overlap and computes the minimum initial interval for an H-SDF graph.

to be created that overlaps different computational iterations as shown in Figure 7(b). To produced this type of a schedule the sample delay characteristic of IP was very important. Because sample-delay in H-SDF represented initial values available to an actor, IP that are downstream from a sample delay could be scheduled before nodes that came before them in strict dataflow. The schedule in Figure 7(b) is the generated schedule for the H-SDF graph shown in Figure 6(a). Notice that in the schedule in Figure 7(b) "CMult1" started before the "nco" even though the dataflow edges in Figure 6(a) seem to require that the "nco" run first.

The IMS algorithm computed a schedule "kernel" that described the repetitive schedule that should be used to continually operate the H-SDF graph properly. An example of this kernel is shown in Figure 7(c). The ability of sample delay to allow IMS to fold long schedules into shorter schedules allowed Ogre to compute the minimum II for a H-SDF graph. This minimum II allowed for maximum throughput on data-flow hardware designs. Once Ogre had completed the scheduling process through IMS, the nodes of the H-

SDF graph were labeled with their start times as shown in Figure 6(b). The schedule kernel produced by IMS allowed control circuitry to be created to control the passage of data through the hardware.

C. Control Synthesis

Once a schedule had been generated for the design, this schedule could be used to generate control circuitry for the system. The Ogre synthesis system used the schedule generated by the IMS algorithm to create a finite state machine (FSM) that controlled when each of the IP in the design was active. By ensuring that IP are active only during their scheduled times, this FSM was able to ensure that data moved between IP during the correct clock cycle.

The FSM generated by the Ogre system assumes that `clock-enable` and `data-valid` signals exist on the IP. Which hardware ports on the IP correspond to these types of signals was described in meta-data. The FSM controlled the flow by properly manipulating the values on the `clock enable` and `data-valid` signals. When the schedule indicated that an IP should start, the FSM would raise the data-valid signal for a single clock, signaling to the IP that it should begin computation because there was real data on its inputs. This `data-valid` signal often was connected to an enable signal on the first bank of pipeline registers in the IP block. In addition to starting the computation with the `data-valid` signal, the FSM raised the `clock-enable` signal on the IP for each of the clock cycles that the IP should be running as determined by the length of the schedule time.

Ogre synthesized the FSM and other interface circuitry in VHDL. When synthesizing the FSM, a VHDL file was produced to implement the proper behavior. The FSM was also added to the Ogre netlist structure as an instance of a component. The FSM was connected to the proper signals in the netlist based on the port names determined from the meta-data. Once the FSM had been connected to the IP in the system, global clock and reset signals were also added and connected to IP as needed.

At this point, the Ogre netlist structure represented a complete and correct hardware design. Synthesizable VHDL was automatically created from the netlist structure. The data path between IP was created and the control from the FSM was connected in VHDL. This generated, top-level VHDL file was then passed to a traditional synthesis flow to create a downloadable bitstream.

V. CONCLUSION

Creation of the synthesis algorithms used in Ogre was possible because of the meta-data in IP-XACT with the extensions describing datatypes and temporal behavior. Meta-data enabled data-flow designs captured in Simulink to be translated into a structural netlist in Ogre. This netlist and the meta-data was then used to create an H-SDF graph that enabled the IMS algorithm to produce a schedule that minimized the II for the design. The IMS schedule was used to synthesize a finite state machine that ensured that data moved through the design

correctly. This FSM was included in a top-level VHDL file that could be synthesized to a bitstream and downloaded to an FPGA.

Ogre demonstrates the ability of meta-data to enable tools to increase design productivity by performing tasks that are traditionally required of human designers when reusing IP. Using Ogre, a designer no longer has to manually create control circuitry to ensure correct flow of data. The designer does not have to worry about bitwidth and datatype correctness. While Ogre may not be suitable for *all* types of designs, the ability of Ogre to synthesize fully functional data-flow designs shows that meta-data can enable tools that can increase design productivity by performing complex architectural synthesis.

REFERENCES

[1] M. McFarland, A. Parker, and R. Camposano, "The high-level synthesis of digital systems," *Proceedings of the IEEE*, vol. 78, no. 2, pp. 301 –318, Feb. 1990.

[2] J. Zhu, "Introduction to c-based high level synthesis," in *ASIC, 2009. ASICON '09. IEEE 8th International Conference on*, October 2009, p. 15.

[3] A. Shatnawi, J. Ghanim, and M. O. Ahmad, "High level synthesis of integrated heterogeneous pipelined processing elements for DSP applications," *Comput. Electr. Eng.*, vol. 30, no. 8, pp. 543–562, 2004. [Online]. Available: http://portal.acm.org/citation.cfm?id=1651877&dl=&coll=GUIDE&CFID=80171792&CFTOKEN=69326439

[4] *IP-XACT Draft/D5: A specification for XML meta-data and tool interfaces*, SPIRIT consortium, 1370 Trancas Street #184, Napa, CA, 94558, May 2009.

[5] A. Arnesen, K. Ellsworth, D. Gibelyou, T. Haroldsen, J. Havican, M. Padilla, B. Nelson, M. Rice, and M. Wirthlin, "Increasing Design Productivity Through Core Reuse, Meta-Data Encapsulation, and Synthesis," in *Proc. of 20th International Conference on Field-Programmable Logic and Applications (FPL 2010)*, September 2010, pp. 538–543.

[6] A. Arnesen, "Increasing Design Productivity for FPGAs Through Intellectual Property Reuse and Meta-Data Encapsulation," Master's thesis, Brigham Young University, April 2011.

[7] E. Lee and D. Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, 1987.

[8] B. R. Rau, "Iterative modulo scheduling," *The International Journal of Parallel Processing*, vol. 24, no. 1, February 1996.