Diplomatic Design Patterns: A TileLink Case Study

Henry Cook
SiFive, Inc.
henry@sifive.com

Wesley Terpstra
SiFive, Inc.
terpstra@sifive.com

Yunsup Lee
SiFive, Inc.
yunsup@sifive.com

ABSTRACT

Modern systems-on-chip (SoCs) incorporate a large and growing number of specialized hardware units that must be integrated into a unified address space via a shared bus topology. This process is labor-intensive and error-prone because the interface requirements of all connected blocks must be mutually satisfied. The design productivity gains derived from the modularity of RISC-V are bottlenecked by the need to integrate the cross product of processor variants, bus ordering behaviors, and slave device capabilities. This growing complexity has stimulated development of new tools and methodologies to enable the completion of complex and parameterized SoC designs. We present two tools used to create correct-by-construction interconnects in the Rocket Chip generator:

Diplomacy is a parameter negotiation framework for generating parameterized protocol implementations. Beyond confirming the mutual compatibility of the system endpoints, Diplomacy enables them to specialize themselves based on knowledge of the other endpoints included in a particular system. TileLink is a highly-parameterized chip-scale shared-memory interconnect standard. The implementation of TileLink in the Rocket chip generator exploits Diplomacy to specialize the interconnect to different levels of protocol conformance.

KEYWORDS

Hardware generators, Agile hardware development, System-on-Chip interconnects

ACM Reference Format:
Henry Cook, Wesley Terpstra, and Yunsup Lee. 2017. Diplomatic Design Patterns: A TileLink Case Study. In *Proceedings of First Workshop on Computer Architecture Research with RISC-V , Boston, MA USA, October 2017 (CARRV'17)*, 7 pages.
https://doi.org/

1 INTRODUCTION

Modern systems-on-chip (SoCs) incorporate an ever-growing number of hardware units specialized to perform particular computational tasks. In order to communicate with one another and memory, these diverse compute resources must be integrated into a shared interconnection network. Such a network typically consists of a hierarchical topology of buses that provide the compute engines with a shared global address space. The process of creating such interconnects is labor-intensive and error-prone because the interface requirements of all connected blocks must be mutually satisfied. The design productivity gains derived from the modularity of RISC-V are bottlenecked by the need to integrate the cross product of processor variants, bus ordering behaviors, and slave device capabilities. This growing complexity has stimulated development of new tools and methodologies to enable the completion of complex and parameterized chip designs [8].

CARRV'17, October 2017, Boston, MA USA
2017. ACM ISBN .
https://doi.org/

We present two tools used to create provably-correct interconnects in the Rocket Chip SoC generator [1]. Our approach is centered around first constructing a graphical model of the properties of the proposed interconnect design, and then using this model to reason as to whether the solution provides all required functionality and will exhibit correct behavior. "Correct" in this case means the generated design will be free from protocol-level deadlock, is guaranteed to make forward progress, and that the cross-product of masters' and slaves' operational requirements is satisfied.

Diplomacy is a framework for negotiating the parameterization of protocol implementations. Given a description of sets of interconnected master and slave devices, and a bus protocol template, Diplomacy cross-checks the requirements of all connected devices, negotiates free parameters, and supplies final parameter bindings to adapters and endpoints for use in their own hardware generation processes. Beyond confirming the mutual compatibility of the system endpoints, Diplomacy enables them to specialize themselves based on knowledge about the capabilities of other endpoints included in a particular system.

TileLink is a highly-parameterized chip-scale shared-memory interconnect protocol standard [5]. The protocol is hierarchically composable and guaranteed to deadlock-free at the transaction level [9]. The implementation of TileLink in the Rocket Chip generator exploits Diplomacy to supply a heterogeneous level of protocol conformance across the interconnect, specialized for the capabilities of devices connected to certain buses.

This paper discusses how various features of diplomatic TileLink have guided the design patterns we have adopted in the Rocket Chip generator. We avoid manually specifying any protocol parameters that can be inferred from a declarative description of the system interconnection network graph. We parameterize our generators to emit hardware based on their view of the rest of the system. We deploy a set of graph transformation patterns that make it easy to re-time links by inserting queues, that comprise thin single-purpose adapters, and that safely create hierarchies of interoperable components.

2 DIPLOMACY

The Rocket Chip generator [1] contains both the internals of the Diplomacy library as well as packages for individual protocol implementations. The sub-generators that comprise Rocket Chip are implemented in Chisel [3], a hardware construction domain-specific language (DSL) that is itself embedded in the Scala language. Chisel provides a set of hardware primitives (e.g., wires, registers, muxes) that can be manipulated using Scala functional programming constructs to efficiently express parameterized hardware designs. Chisel

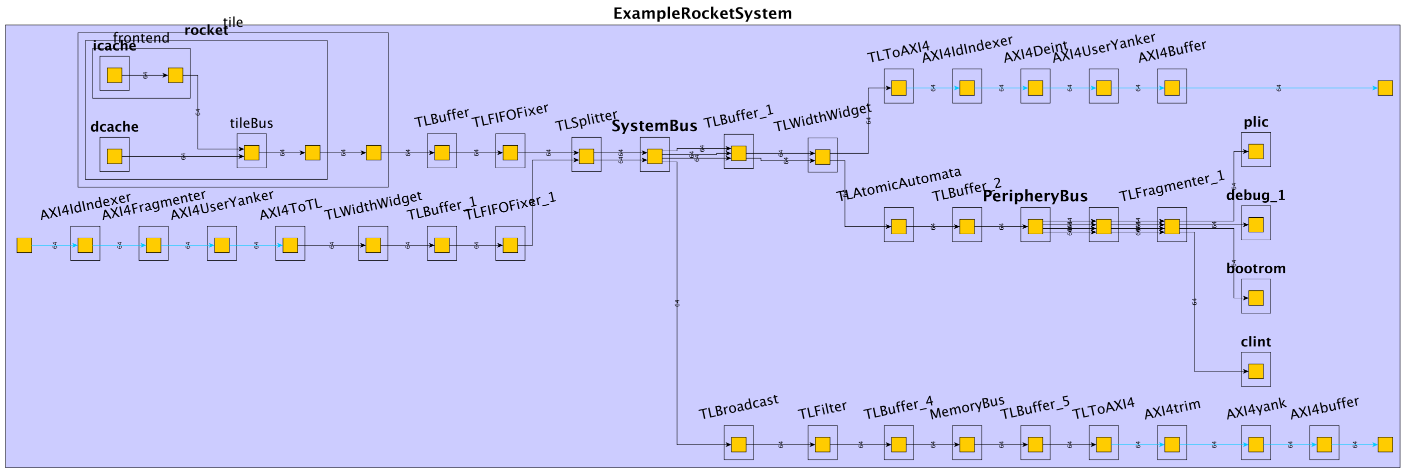

Figure 1: Example of the diplomatic graph created by the Rocket Chip generator. The yellow squares are diplomatic nodes and arrows are diplomatic edges. Note that two colors of edge indicate that both TileLink and AXI4 protocols are deployed in this SoC interconnect.

designs can be compiled into structural RTL languages such as Verilog. While Chisel makes it easy to write parameterized design components, it provides no particular functionality related to parameterization other than that built in to Scala itself.

Diplomacy extends Chisel, being a library that provides a parameter negotiation framework for generating hardware connected to a shared interconnection network. Operating on a description of a SoC design expressed as a directed graph of interconnected nodes, Diplomacy cross-checks a set of protocol-specific requirements over relevant subsets of the interconnect. Diplomacy allows for the negotiation of any free parameters within a protocol, such as particular data or control wire widths, customizing them based on the topology expressed by each sub-graph of communicating nodes. After enabling all the endpoint nodes to negotiate protocol parameter values, Diplomacy then supplies concrete parameter bindings to individual adapters and endpoint generators, which can use them to drive their own individual hardware generation processes.

Diplomacy is thus an example of two-phase hardware elaboration. The first phase is parameter negotiation, wherein the topology of the graph is discovered and the nodes negotiate the value of the parameters on every edge. The second phase is concrete module generation, in which the Chisel compiler is invoked on the module hierarchy associated with the node graph. As each Chisel module is elaborated, it can make use of the diplomatic parameters that have been precomputed by its associated diplomatic nodes.

The fundamental abstraction used by Diplomacy is a graph of nodes and edges representing the topology of the shared interconnect. Nodes represent points in the design where diplomatic parameters are used to emit hardware. Edges represent a directed pairing of master and slave interfaces, where the *source* node presents a master interface and *sink* node presents the matching slave interface. Nodes may participate in multiple edge pairings, and may either have a single type of interface (an endpoint node), or may forward from one type to the other (an adapter node). Edges communicate a protocol-specific set of parameters between masters and slaves, and these parameters are specified to flow either *outward* from source to sinks, or *inward* from sinks to sources. Edges are used to elaborate the wires that actually create the physical connections in the final design, in the form of wires or module IOs. A module may have multiple nodes and a node may have multiple edges.

Beyond confirming the mutual compatibility of the system endpoints, Diplomacy enables them to specialize themselves based on knowledge of the other endpoints included in a particular system. Each node supplies its associated generator with a set of edges that contain a view of the interconnect as seen from that node. By accessing the parameter values available on a particular edge, the generator can specialize its behavior according to the capabilities of that edge. For example, it may size the hardware it generates internally according to the negotiated width parameters supplied by the edge.

Protocol parameters may be independently specified by nodes or derived from negotiations between nodes. Some independent parameters are supplied by source node endpoints, and others by sink node endpoints. Adapter nodes may perform some transformation on the parameters passed through them in either direction. Examples of parameters include:

- The cardinality of sources connected to a particular sink.
- The cardinality of sinks connected to a particular source.
- The type and size of operations issued by each master.
- The type and size of operations issued by each slave.
- The type and size of operations allowed on a particular range of addresses.
- Other properties governing allowed behavior on particular address space regions (e.g., executability, cacheability).
- Ordering requirements on operations over edges (e.g., FIFO).
- Presence of certain fields within control wire bundles.
- Widths of fields within control or data wire bundles.

Parameter negotiation itself consists of two independent subprocesses. Beginning with the source endpoint nodes, some parameters flow outwards until they have reached all sink nodes. Independently, beginning with the sink endpoint nodes, other parameters flow inwards until they have reached all source nodes. Thus, every edge receives a complete set of parameters describing it in both directions.

Negotiation may fail if a certain adapter transformation has requirements that cannot be met, or if a certain endpoint is required to exhibit behavior which it cannot implement. Such a failure is desirable from the perspective of designer productivity because it occurs so early in the design-verify feedback loop [8]. Rather than having to detect that a interconnect has been misconfigured through laborious simulation of transactions over the bus, designers are notified of issues before the hardware has even been generated.

Diplomacy is bus-protocol-independent, in that any protocol with parameterized features can be templated using Diplomatic primitives and deployed alongside other Diplomatic protocol implementations. Modules that serve as converters between the bus protocols also translate their parameters between each type of diplomatic nodes, allowing the entire interconnect to be composed of heterogeneous protocols while still ensuring overall correctness. In addition to developing TileLink, our own diplomatic shared-memory coherence protocol discussed in the following section, we have templated multiple AMBA protocols [2] (e.g., AXI4, AHB-Lite, APB) in order to allow them to be expressed and deployed diplomatically.

Figure 1 shows a simple example Rocket Chip design's diplomatic graph. The yellow squares are diplomatic nodes and arrows are diplomatic edges. Note that two colors of edge indicate that both TileLink and AXI4 protocols are deployed in this SoC interconnect: AXI4 is used to communicate with the outside world and TileLink is used for internal connectivity. The upper left collection of nodes is a Rocket processor with its instruction and data caches. The lower left series of nodes is an AXI4-to-TileLink bridge. The center right sequence of nodes are various peripheral devices including a boot ROM and debug unit. They are bracketed by two TileLink-to-AXI4 bridges for issuing cacheable and uncacheable memory access operations respectively.

3 TILELINK

TileLink [7] is a chip-scale interconnect protocol standard for providing multiple processing elements with coherent access to shared memory and other memory-mapped devices. Specifically, TileLink is designed to be deployed in a System-on-Chip (SoC) to connect general-purpose multiprocessors, co-processors, accelerators, caches, DMA engines, memory controller, and simple or complex peripheral devices. The protocol is optimized to be efficient when deployed within tightly-coupled, low-latency SoC buses. It can also be implemented over hierarchically-composable, point-to-point networks, and can scale down to interface with low-throughput slave devices or scale up to provide high-throughput interconnects. TileLink provides coherent access for an arbitrary mix of caching or non-caching masters to a physically-addressed, shared-memory system. Cache coherence is maintained by a customizable, MOESI-equivalent protocol based on hierarchical composition [4, 11]. The protocol supports out-of-order transaction completion to improve throughput for concurrent operations.

The TileLink specification maps closely onto the abstractions used by Diplomacy [5]. TileLink agents are semantically equivalent to diplomatic graph nodes. TileLink links are semantically equivalent to diplomatic graph edges. Figure 2 provides an overview of an example TileLink network topology graph.

Within a link, TileLink contains five logical channels, which correspond to the priorities of the messages that they carry. Each channel consists of control and data signals that are transmitted using a decoupled, ready-valid based handshaking protocol. TileLink memory operations comprise a series of messages sent over channels, obeying certain transaction rules (e.g. all requests have responses). To avoid deadlock, TileLink specifies a priority amongst the channels' messages that must be strictly enforced. The prioritization of messages across channels is A « B « C « D « E, in order of increasing priority. Channels are directional, in that each passes messages either from master to slave interface or from slave to master interface. Figure 3 illustrates the directionality of the five channels.

The two basic channels required to perform memory access operations are:

Channel A. Transmits a request that an operation be performed on a specified address range, accessing or caching the data.

Channel D. Transmits a data response or acknowledgement message to the original requestor.

The protocol conformance level that enables the system to include coherent caches adds three additional channels, which provide the capability to manage read/write permissions on cached blocks of data:

Channel B. Transmits a request that an operation be performed at an address cached by a master agent, accessing or writing back that cached data.

Channel C. Transmits a data or acknowledgment message in response to a Channel B request. Also writes back dirtied cached data.

Channel E. Transmits a final acknowledgment of a cache block transfer from the original requestor, used for serialization.

The design of TileLink is intended to exploit the features of Diplomacy, in that the protocol is modular with respect to the type of operations that will be transmitted over a given link. Master and slave agents can agree that certain operations will not be performed (perhaps removing the need for entire physical channels to be elaborated), or that there is a maximum size of messages that will be sent (removing the need for beat-counting control logic). Fields of unused message types can be pruned from the signal encoding, and identification fields can be expanded only for portions of the network graph that require additional bits to route requests and responses. While the TileLink specification [7] explicitly enumerates three levels of protocol conformance, it does not preclude implementations from adopting their own subset of the protocol, and the Rocket Chip implementation of TileLink uses Diplomacy to subset different features at a very fine granularity.

Additionally, tracking and negotiating which address spaces belong to which region types as part of the first phase of diplomatic

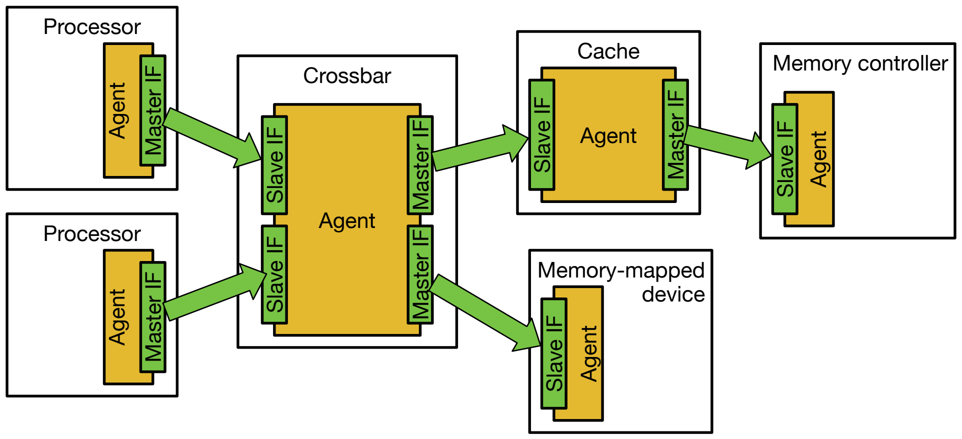

Figure 2: Example of a TileLink network topology DAG with four endpoints. The yellow agents are diplomatic nodes and the green links are diplomatic edges. Two modules contain an agent that has both a master and a slave interface, i.e. diplomatic adapters.

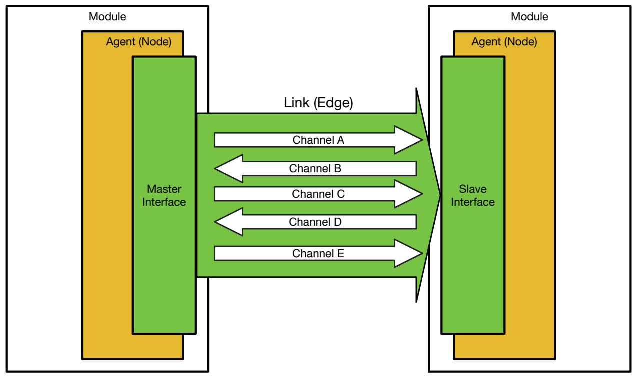

Figure 3: The five channels that comprise a TileLink link/edge between any pair of agents/nodes.

elaboration allows us to elide any "protection" or "region" fields within the messages themselves. The legality of memory accesses must be determined by the sender before the transaction is initiated. While we do include an error field for dynamically reporting transient faults, the availability of statically knowable properties of the memory system via Diplomacy means that we do not need to devote actual hardware resources to transmit message fields that would be present solely for determining region type compatibility.

TileLink provides formally-verifiable deadlock freedom for any SoC consisting of compliant network and endpoint implementations [9]. This property emerges from the following protocol design decisions:

Generality of protocol interfaces: All agents use the same transaction structure [4, 11], though the message fields themselves might be parameterized differently

Scalability rules for hierarchy: The topology of the interconnect must be a Directed Acyclic Graph (DAG). Preventing cycles in the graph is necessary to avoid deadlocks based on cyclic hold-and-wait behavior.

Strict prioritization of channels: Messages transmitted on a higher priority channel cannot be blocked by messages on a lower priority one.

Forward progress through decoupled interfaces: We provide rules governing when message recipients are allowed to reject a proffered message. It can be rejected only for a bounded time period, or while waiting on a higher priority message to be sent or responded to.

The above properties ensure that TileLink is hierarchically composable, allowing for highly-configurable network topologies that mix and match components with stateless bus-width adaptation and burst fragmentation, and ease register-stage insertion for retiming links. Zero-cycle response time is not only legal but support for it is mandatory, and legal combinational couplings of ready/valid signals are also specified. The design ramifications of these features will be discussed in more detail in the following section.

4 DESIGN PATTERNS

In this section we will examine how Diplomacy and TileLink have enabled us to use a set of hardware design patterns that are well-adapted to a generator-centric development environment.

4.1 DRYing Out Parameterization

Don't Repeat Yourself (DRY) is a software development principle stated as "Every piece of knowledge must have a single, unambiguous, authoritative representation within a system" [6]. One of the main productivity advantages of hardware generators over traditional HDL approaches is that many similar designs can be captured and expressed as parameterizations of a single generator, which is a boon when making changes that would otherwise require manual updates to each individual design. However, while parameterized generation is an excellent source of DRYness across a family of hardware designs, the parameters themselves can rapidly become a source of non-orthogonalized complexity.

Consider the example of a parameterized arbiter generator that produces arbiters that take N inputs, mux them down to a single output, and append a log2(N) integer to the source field of arbitrated messages for use in routing their responses to the correct port. In a non-diplomatic code base, we might have to bind the cardinality N in several places: the constructor to the Arbiter module and the constructors of all downstream expanded-source-field wires. Furthermore, we must then actually instantiate and wire up as many inputs as we previously declared that the crossbar is going to have. This proliferation of the number of places that N must be specified results in drag on reconfigurability. All the different parameters that are based on N must agree on N's value, and if this agreement is not checked, misconfigurations will occur as the design space evolves.

Diplomacy allows us to avoid specifying any parameters that can instead be inferred from the declarative description of the graph. In the case of a diplomatic arbiter generator, the fact that we connected N source nodes to the arbiter's node is a single, unambiguous, authoritative representation. We do not have to explicitly state the cardinality of N anywhere in the code base, and the value of N that is inferred from the graph topology during the first phase of elaboration is automatically propagated to both the arbiter generator as well as to all downstream edges used to parameterize wire widths during the second phase.

4.2 Hardware Generation with A View

Beyond serving as an unambiguous source of cardinality information, the diplomatic graph provides generators with knowledge of the capabilities of the other nodes in the system with which they are interconnected.

Master nodes can see the capabilities of all slaves whose memory-mapped addresses are visible to them. This view can be used to determine whether a particular memory operation will be legal to issue, either at Chisel compile time, or within the generated hardware control logic. The TLB in Rocket Chip's L1 cache generator is an example of such a view-exploiting master device. It calculates whether a given page supports Read/Write/Execute operations, whether it can be safely cached, or whether it has support for certain atomic operations, all based on the diplomatically-generated map of addresses to RegionTypes. Statically, at Chisel compile time, the TLB checks whether the size of caching memory transfers supported for each slave device includes the the size of that operation issuable by the containing processor's cache. The TLB also generates hardware circuits that calculate the permissions to be recorded in a particular page table entry, again based in-part on diplomatic information made available from the graph.

Conversely, slaves can see the capabilities of all masters whose operations can reach them. For example, an L2 cache could see both the total number of masters that can possibly send it requests, which would be useful for generating the size of MSHR buffers. Additionally, it could count the total number of masters that have the capability of caching its data, which would be useful for generating the number of directory bits needed to track the locations of shared copies of a block. Finally, it could track the ordering requirements of particular masters, which would be necessary to properly generate control logic for issuing out-of-order requests to the outer memory system.

In a similar fashion, adapter nodes provide interconnect generators with information about the cardinality and capabilities of both masters and slaves. The TileLink protocol only requires the use of all five channels when some master is capable of caching blocks of data. If an interconnect generator creates separate physical networks for each channel (a reasonable choice for obtaining TileLink's required inter-channel prioritization), it can elide the wiring hardware for the unneeded channels when it determines that no masters can issue cache block transfers over itself.

Overall, this approach to hardware generation encourages designers to take a much more global view of the system, even when defining the generative behavior of localized components. The parameters described via diplomatic edges form a stable API with which a generator can query the surroundings in which it instantiated, no matter where in the global design that may be.

4.3 Correct By Composition

The properties of the the TileLink protocol enumerated in the previous section have the advantage of ensuring that design changes encompassing certain transformations on an extant diplomatic graph are guaranteed to produce another correct graph and implementation. These transformations are all based on injecting new sub-graphs that preserve the DAG property. In this subsection we discuss three particular transformations that have shaped the structure of the Rocket Chip codebase.

We term the first of these transformation patterns *combinational composition*. In this pattern, multiple adapter nodes are composed in a linear sequence. This pattern is enabled by the flow control rules of the TileLink protocol: responses to messages may be sent on the same cycle a request is received, and the ready/valid signals of each channel may be combinatorially forwarded from the inward side of the adapter to the outward side and vice versa.

Taken together, these combinational flow control properties encourage the use of "thin" adapters, where every adapter serves a unique and orthogonalized purpose. For example, we provide individual TileLink adapters that

- modify control signals (e.g., fragment burst messages into a series of single-beat messages)
- adjust message field widths (e.g., widen the width of the data bus plane)
- manage transaction-level requirements across messages (e.g., ensure that a series of messages request and responses obey FIFO ordering)

Larger logical adapters can be created by composing multiple small adapters to achieve a complete functionality, while the individual adapters can be unit tested and verified independently of one another. This pattern is well-displayed by the chains of adapters converting from TileLink to AXI4 or vice versa in Figure 1. Beyond the productivity gains in verification methodology, this pattern allows us to create standardized bus attachment points to which authors of new peripheral devices or co-processors can attach their contribution. Each attachment point comprises of a set of adapters providing well-defined behaviors with zero cycle delay overhead.

The second transformation pattern, *sequential composition*, additionally exploits the fact that the decoupled nature of the TileLink channels make it trivial to insert an interstitial adapter that is a queue. We provide a standardized set of buffering parameters that control not only the depth of the queue on each TileLink channel, but also the flow control parameters (e.g., pipeling enqueue of back-to-back messages or flowing messages through empty queues). Making the insertion of queues along chains of nodes trivial makes the design process of searching for optimal timing decoupling points very lightweight. This flexibility in turn enables faster iterations with backend QoR teams.

The final transformation pattern, *hierarchical composition*, is the most general and the most powerful. Enabled by TileLink's deadlock freedom guarantees, we can swap out any node of the diplomatic graph for an entirely new subgraph that maintains the same properties at its inward and/or outward attachment points. For example, this pattern is a natural fit for creating hierarchies of inclusive caches, since the branching factor of each layer of caches is orthogonal and invisible to the masters on the inward side of the innermost cache's node. In a similar fashion, a single adapter node can be replaced with an address filter adapter node and banked copies of that adapter backed by unique memory channels.

Declaring that a node of some type exists and can be attached to by generators with certain properties, without stating a priori how many such generators must be attached to it, unlocks a powerful capability for system reconfigurability via modular mix-in composition [10]. Scala's (and thereby Chisel's) support for multiple inheritance allows us to compose sub-graph components via traits that statically depend only on certain nodes being available for connection. Figure 4 provides an example of this pattern, wherein traits describing the connection between an abstract set of processors and memory are combined with traits that supply a concrete coherence manager and a concrete set of processors. Incompatibilities can be detected at Scala compile time or during either phase of diplomatic Chisel elaboration.

```scala
// Trait containing attachment points common to all systems
trait ConnectsToMemory {
  val processorMasterNode: TLSourceNode // abstract member
  val memorySlaveNode: TLSinkNode = LazyModule(new TLRAM).node
}

// No coherence manager provided, only a single cache is safe
trait ConnectsIncoherently extends ConnectsToMemory {
   require(processorMasterNode.masters.size <= 1)
   memorySlaveNode := processorMasterNode
}

// Make some coherence manager and insert it in the node graph:
trait ConnectsViaBroadcastHub extends ConnectsToMemory {
  val hub = LazyModule(new TLBroadcastHub)
  hub.node := processorMasterNode
  memorySlaveNode := hub.node
}

trait ConnectsViaL2Cache extends ConnectsToMemory {
  val l2 = LazyModule(new TLCache)
  l2.node := processorMasterNode
  memorySlaveNode := l2.node
}

// Add some master nodes and reify the attachment point:
trait HasOneCore extends ConnectsToMemory {
  val core = LazyModule(new Rocket)
  val processorMasterNode = core.node
}

trait HasTwoCores extends ConnectsToMemory {
  val cores = List(2).fill(LazyModule(new Rocket))
  val xbar = LazyModule(new TLXbar)
  val processorMasterNode = xbar.node
  cores.foreach { c => xbar.node := c.node }
}

// Compile and elaborate correct hardware:
class SingleCoreSystem extends HasOneCore  with ConnectsIncoherently
class DualCoreSystem   extends HasTwoCores with ConnectsViaBroadcastHub
class DualCoreL2System extends HasTwoCores with ConnectsViaL2Cache

// Fails at Scala compile time due to missing TLSourceNode instance:
class IncompleteSystem extends ConnectsViaL2Cache

// Fails during elaboration due to failed requirement:
class UnsafeSystem     extends HasTwoCores ConnectsIncoherently
```

Figure 4: Using multiple inheritance to inject different node hierarchies at two places in the memory subsystem.

5 CONCLUSION

In this paper we have discussed how various features of diplomatic TileLink have guided the design patterns adopted in the Rocket Chip generator. We avoid re-specifying any parameters that can be inferred from a declarative description of the system interconnect graph. We parameterize our generators to emit hardware based on a diplomatically negotiated view of the system. We deploy a set of safe graph transformation patterns that make it easy to re-time links with queues, that compose more complicated adapters from single-purpose ones, and that inject hierarchies of composable components at standardized attachment points.

We see many forthcoming opportunities to deploy correct-by-construction software development methods to other aspects of SoC hardware component integration. Clock domains are one area where each mixed-in component could specify how it relates to the overall interconnect graph, and the correct clock and reset would be supplied to the second phase elaboration automatically. Interrupts are another type of signal where we would like to avoid any explicit notion of cardinality and instead infer it from graphical structure. Finally, we are working to develop endpoint generators that are protocol-independent, presenting generator authors with standard APIs that allow their device to be deployed regardless of the underlying interconnect protocol.

The Rocket Chip code base containing Diplomacy itself, all of our diplomatic bus protocol implementations, and a variety of TileLink-compatible cache, interconnect, and device generators, is open source and available as part of the Free Chips Project at https://github.com/freechipsproject/rocket-chip.

REFERENCES

[1] Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, et al. 2016. The rocket chip generator. (2016).

[2] ARM AMBA AXI and ACE Protocol Specification AXI. 2011. *AXI4, and AXI4-Lite, ACE and ACE-Lite*. Technical Report. Technical report.

[3] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. 2012. Chisel: constructing hardware in a scala embedded language. In *Proceedings of the 49th Annual Design Automation Conference*. ACM, 1216–1225.

[4] Jesse G Beu, Michael C Rosier, and Thomas M Conte. 2011. Manager-client pairing: a framework for implementing coherence hierarchies. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 226–236.

[5] Henry Michael Cook. 2016. *Productive design of extensible on-chip memory hierarchies*. University of California, Berkeley.

[6] Andrew Hunt and David Thomas. 2000. The Pragmatic Programmer. *Addison Wesley* 15 (2000).

[7] SiFive Inc. 2017. *SiFive TileLink Specification*. Technical Report. SiFive, Inc. https://www.sifive.com/documentation/tilelink/tilelink-spec/

[8] Yunsup Lee, Andrew Waterman, Henry Cook, Brian Zimmer, Ben Keller, Alberto Puggelli, Jaehwa Kwak, Ruzica Jevtic, Stevo Bailey, Milovan Blagojevic, et al. 2016. An agile approach to building risc-v microprocessors. *IEEE Micro* 36, 2 (2016), 8–20.

[9] Kenneth McMillan. 2016. Modular specification and verification of a cache-coherent interface. In *Proceedings of the 16th Conference on Formal Methods in Computer-Aided Design*. FMCAD Inc, 109–116.

[10] Martin Odersky and Matthias Zenger. 2005. Scalable component abstractions. In *ACM Sigplan Notices*, Vol. 40. ACM, 41–57.

[11] Meng Zhang, Alvin R Lebeck, and Daniel J Sorin. 2010. Fractal coherence: Scalably verifiable cache coherence. In *Microarchitecture (MICRO), 2010 43rd Annual IEEE/ACM International Symposium on*. IEEE, 471–482.