1 Basic

Since the introduction of Verilog in 1984, hardware description languages have become the dominant technique used for designing hardware logic. Over time, features have been borrowed from the software language community, increasing their descriptive ability. Sadly, proving hardware designs correct before fabrication remains persistently out of reach; large scale reasoning about highly parallel algorithms remains problematic. This leads design teams to spend a large fraction of development budget on validation, preventing agile exploration of new ideas and design evolution.

To address these issues, there has been a long term research effort into Guarded Atomic Actions (GAA), structuring and automating the parallel interactions permissible between unitary sections of logic. By generating code that is guaranteed to be sequentially consistent and by automating the propagation of execution guards, designs become more tractable at design time, leading to correctness proofs of designs rather than merely relying on validation.

Unfortunately, these ideas have not spread widely into the hardware design world for a variety of reasons:

— The only complete implementation is a proprietary licensed product

— The constructs are embedded in a hybrid of a functional language with aspects taken from HDLs, unfamiliar to most practioners

— The generated verilog is quite difficult to read The volume of the generated code has made verification even more difficult

AtomicC attempts to address all these issues, providing a structured alternative to Verilog that is quick to understand and easy to deploy. By making the source code openly available to the research community, we hope that broader analysis of the concepts of GAA will become practical, leading to methods that better control the spiraling costs of hardware design.

1.1 Introduction

AtomicC is a timed, structural hardware description language for the high level specification of algorithms to be instantiated directly in hardware. AtomicC extends C++ with support for Guarded Atomic Actions [1, 2, 3]: Bluespec-style[4] modules, rules, interfaces, and methods.

The language is designed for the construction of **modules**[5] that are *correct-by-construction composable*: validated smaller modules can be aggregated to form a larger validated module with no loss of correctness of the component modules:

— Module interactions are performed with latency insensitive[6, 7] **method** calls, allowing methods to enforce invocation pre-conditions and transitive support for stalling.

— Module behaviorial statements are encapsulated into transactions (**rules**) following ACID semantics [8, 9]:

  — *Atomic*: all enabled rules in all modules execute on every clock cycle.

  — *Consistent*: The compiler synthesizes control signals, allowing rules to fire only when their referenced method invocations (*implicit conditions*) are ready.

  - *Isolated*: all rules executed during a given clock cycle are *sequentially consistent* (SC) [10], guaranteeing each rule executes independently of any other rules executing at the same time [9, Sec. 7.1].
  - *Durable*: all transactions read from and write to state elements in the design
- An **interface** is a named collection of method signatures, defining the behavior of an abstract data type(ADT) [11]. Modules can declare multiple **interfaces**, giving each interface an explicit name, giving flexibility in coupling with other modules. Interfaces can be exported (defined in the module) or imported (used in the module, but defined externally), giving flexibility in algorithm representation [12, Sec. 4.1]. By using interfaces to describe behavior of a module instead of relying on inheritance, the user is able to specify clearly that no implementation information is inherited.
- All state elements in the hardware netlist are explicit in the source code of the design. All module data is private to the module, accessable externally only by method invocation.

These features support the reliable reuse of pre-compiled, incrementally validated libraries, improving productivity on large designs.

AtomicC does not attempt to emulate the behavior of all C++ constructs in hardware. It instead uses a subset of the C++ language to specify code blocks that have the property that there is at most one assignment to each state element. This form is called *static single assignment* form, or SSA form[13] and is a functional program[14]. In the eventual runtime execution, these assignments are all made in a single clock cycle, when the rule or method is enabled.

To preserve the standard interpretation of C++ source code in methods and rules, a **modification in-private** [15, Sec. 3.2] execution model is used:

1. Wrap each rule/method with prelude code and postprocessing code
2. Prelude code: For each state element **A** in module, add the declaration of a shadow item:

   ```
   decltype(this->A) A = this->A;     // create shadow of state element
   ```

3. Postprocessing code: For each state element actually written during execution of the code block:

   ```
   this->A = A;                  // update state elements only at end of code block
   ```

   All assignments in the postprocessing section occur on a single clock cycle.

The AtomicC compiler generates a single Verilog module for each defined AtomicC module. Existing Verilog modules can be called from and can call AtomicC generated modules. Standard Verilog backend tools are used to synthesize the resulting ASIC or FPGA. Particular emphasis is put on making the Verilog output both readable and stable to incremental change. Synthesized names in the verilog output depend only on the named structual context, not on aspects like 'source line number' that will change globally, even when there is no local module definition source change. Incremental source code changes produce locally incremental generated code changes, easing the management of ECOs and version control on successive releases (for example, management of Verilog output files using git repositories). Since project development often involves many people over an extended interval of time, predictable reuse and evolution of modules is considered key to improving productivity.

Like Connectal[16], AtomicC designs may include both hardware and software components, using interfaces to specify hardware/software communication in a type safe manner. The

AtomicC compiler generates the code and transactors to pass arguments between hardware and software.

In summary, although AtomicC is a way of constructing highly parallel hardware, the usage model is quite simple:

— Methods/rules are written as though they were locally serially executed,

— Updates to state elements only occur atomiccally, as transactions. Intermediate state evolution is not visible to other transactions,

— Concurrent execution of transactions is guaranteed to be SC by the compiler framework,

— Correct concurrent implementation of the code for each individual method/rule is guaranteed by the compiler's translation of the SSA source text into logic.

Of course, if the underlying algorithm is not designed to allow parallel execution of incremental computations, it will perform poorly and there is nothing the compiler can do to help. AtomicC allows the user to focus solely on the algorithm itself, without the burden of the bare mechanics of orchestrating concurrent consistency, increasing quality and productivity.

1.2 Modules

The basic building block of AtomicC is the module declaration, made of 3 parts:

— Instantiation of state elements used by the module,

— Interface declarations for interacting with other modules,

— Rules, which group assignment statements and method invocations into atomic transactions.

There are 2 types of methods:

— **Value method functions** allow *inspection* of module state elements.

— **Action method procedures** allow *modification* (including inspection) of state elements, can take parameters and do not have return values. A compiler generated **valid** signal indicates that the caller wishes to perform the method invocation.

Both value and action methods use a compiler generated **ready** signal to indicate when the callee is available and stall scheduling of the calling transaction until execution pre-conditions are statisfied.

AtomicC uses **valid/ready** *hand-shaking signalling* [17, 18] to invoke action methods, giving both the invoker(master) and invokee(slave) the ability to control invocation execution timing. The master uses the **valid** signal of an action method to show when parameter data is available and the operation should be performed. The method invocation succeeds only when both **valid** and **ready** are HIGH in the same clock cycle.
In TRS notation[1, p. 22]:

$$\pi(M_i) \equiv ready(M_i) \ \wedge valid(M_i).$$

1.3 Scheduling

1.3.1 Goals

In software systems, to guarantee *isolation* in the presence of parallelism, *dynamic allocation*[9, p. 377] of schedules and locking[19, Sec. 11.2] are used. In hardware design with AtomicC, the set of state elements accessed by a transaction, the operations on these state elements

(read-only or write) and the boolean condition when the transaction is performed are all known at compile time. This allows *static allocation*[9, Sec. 7.3.1] of **schedules** (sequences of transaction execution) and compile time validation of SC.

1.3.2 Algorithm

The scheduling algorithm is:

- For each module, rules and methods that overlap usage of state elements (*read set* and *write set*[19, Sec. 10.1.2] [20]) are greedily gathered into *schedule sets*. Since there can be no execution interactions between sets, each set will be independently scheduled.
- A *constraint graph* is a partially-ordered digraph modeling the schedule sequencing dependencies within a *schedule set*:
  - *nodes* in a constraint graph represent atomic rule and method instances,
  - *edges* represent **write-after-read (WAR)** ordering dependency for a specific storage element[21, Sec. 3].

    In addition, each edge has a symbolic boolean *edge condition* for when the the ordering dependency exists: the boolean condition when one rule/method actually reads a given state element and the other actually writes it.
- The transitive closure of these orders on the constraint graph nodes dictate the **schedule** in which each rule must *appear* to execute in order to be considered SC [19, Sec. 11.1]. Of course, since all rules execute in a single cycle, "schedule" does not refer to an actual time sequenced evolution of state, but to a *conceptual* "sub-cycle" ordering.
- For each pair of nodes in the constraint digraph, we define the *node condition* between 2 nodes as the conjunction of the *edge conditions* of all the edges between them (i.e., the condition that *any* of the edges causes a dependency). For each cycle in the digraph, we define the *path condition* as the disjunction of the *node conditions* for all sequential pairs of nodes in the cycle (i.e., the condition that *all* the edges, hence the cycle exists).
- Since potential conflicts between methods (called from rules outside the module) and module rules are quite common, if cycle has some method M & some rule R, then the compiler can rewrite the term $valid(R)$ to add a disjunction with the term $\neg valid(M)$, breaking the cycle.
- When the *path condition* is not identically false, a total ordering of the digraph can not be guaranteed and the *schedule set* is not SC. In this case, the compiler or linker reports an error, requiring resolution by the user.

A simple example of a constraint graph is given in Annex B, at the end of this document.

Since AtomicC performs scheduling analysis independantly for each declared module, method invocation conflicts in rules cannot be validated. Schedule processing for rule method calls is delayed until the "module group binding" stage of linking, where separately compiled AtomicC output is combined and verified for SC scheduling. Errors and conflicts detected at this stage must be repaired in the module source text and recompiled before proceeding.

1.3.3 Previous scheduling work

In Rule Composition[3], scheduling is reformulated in terms of rule composition, leading to a succinct discussion of issues involved, including a concise description of the Esposito

and Performance Guarantees schedulers. The resulting schedules are quite close to the user-specified scheduling in AtomicC. In contrast to AtomicC, the Bluespec kernel language they use for analysis also has a sequential composition operator, creating rules that execute for multiple clock cycles.

The Esposito Scheduler[22, 3], is the standard scheduler generation algorithm in the Bluespec Compiler. It uses a heuristic designed to produce a concrete total ordering of rules.

The Performance Guarantees scheduler[23] was proposed to address issues with intra-cycle data passing.

1.4 Compilation

In traditional software, program startup at runtime allocates and populates data structures used during execution. Although dynamic creation of state elements is not possible for hardware, we need the ability to create them programmatically, improving efficiency of the design creation process. The compilation of an AtomicC program is similar to that of C++, but adds an *elaboration* phase [24, Sec. 1][25, 26, 27], translating the source parse tree into a high-level intermediate *abstract syntax*. This is quite similar to the approach suggested by Bluespec-2[28].

During *static elaboration*, constructors are executed for statically declared data elements, allowing allocation of new instances of modules and parameterized rule creation. Any C++ constructs may be used, but the resulting netlist must only contain synthesizeable components.

AtomicC execution consists of the following phases:

- *compilation*
  - Parsing,semantic checks,
  - Static elaboration,
  - Translation to an intermediate representation (IR),
  - Verilog netlist generation from the IR,
- *Linking*: Modules across the project are incrementally compiled and unit tested. As modules are reused in varying contexts, it is necessary to validate if restrictions need to be added for concurrent calls to conflicting methods in the module interface. This checking is done by a *linker*, ensuring that rule/method access across a set of Verilog output is free of inter-module schedule conflicts,
- *Logic synthesis, physical backend processing.* This is performed with existing backend tool flows
- *Formal verification* using the Coq Proof Assistant, etc.
- *Verification*: performed using existing backend tooling,
- *Hardware execution.*

1.5 Future work

Need to describe multi-cycle rules and pipelining.

Need to have a way to support sequencing of operations

Need to have a way to support model checking (say 'module B is a behavioral description of module A') Show example with diff eqn solver from Sharp thesis.

C block semantics do not correctly process the 2 statements: a = b; b = a;. (binding of read values should occur at beginning of block, so that it is clear the 2nd assign refers to the 'previous' value). Thinking again: if we retain C semantics, we have: temp = a; a = b; b = temp;, which gives the correct value mapping.

Multiple clock domains

Need reference definition for 'static elaboration'. from Newton: Static elaboration does not change the semantics (types, evaluation rules) of the Regiment language, it merely opportunistically pushes evaluation forward into compile time

Coding FSMs as 'case' statements in Verilog (to integrate with verification tools).

Physical partitioning is used to separate design into separately synthesized pieces, connected using "long distance" signalling. Parallel synthesis; bitstreams combined.

2 Classes

2.1 Module declaration and definition

A module, defined using the keyword "___module", results in the generation of a corresponding verilog module in the compilation output file. It includes local state elements, interfaces exported, interfaces imported and rules for clustering operations into atomic transactions.

Modules are independently compiled, even if they exist in the same compilation unit. Rule and interface method scheduling logic is generated as part of the generated module. Scheduling constraints (read set, write set and relation to other scheduled elements) are generated into a metadata file, allowing schedule consistency between modules to be verified by the linker.

[*Example*:

```
__module Echo {
    EchoRequest      request;              // exported interface (defined by this module)
    EchoIndication   *indication;          // imported interface (defined by the instantiator of this module)
    bool busy;
    __int(32) itemSay;
    ...
    // implementation of method request.say(). Note the guard "if (!busy)".
    void request.say(__int(32) v) if(!busy) {
        itemSay = v;
        ...
    }
    void request.saw(__int(16) a, __int(16) b) if(!busy) {
        ...
    }
};
```

— *end example*]

To reference a module from a separate compilation unit, use "___emodule". External module definitions need only specify the exported/imported interfaces.

[*Example*:

```
__module EchoResponder {
    EchoIndication   indication;          // exported interface
};
```

— *end example*]

2.2 Module interface definition

An AtomicC interface is essentially an abstract class similar to a Java interface. All the methods are virtual and no default implementations are provided. AtomicC style uses composition of interfaces (using ___connect) rather than inheritance.

The ___interface keyword defines a list of methods that are exposed from an object that can be composed as a unit. Instead of using object inheritance to define reusable interfaces, they are defined/exported explicitly by objects, allowing fine-grained specification of interface method visibility.

Methods of a module are translated to value ports for passing the method arguments and a pair of handshaking ports used for scheduling method invocations.

References to an object can only be done through interface methods. State element declarations inside an object (member variables) are private.

[*Example*:

```
__interface EchoRequest {
    void say(__int(32) v);
    void say2(__int(16) a, __int(16) b);
};
```

— *end example*]

2.3 Guard clauses on module interface methods

1 Method definitions in ___module declarations have the form:

atomicc-method-definition:
decl-specifier-seq$_{opt}$ *interface-qualifier-seq identifier parameters-and-qualifiers function-body*

interface-qualifier:
identifier `.`

interface-qualifier-seq:
interface-qualifier
interface-qualifier-seq interface-qualifier

atomicc-function-body:
if-guard$_{opt}$ *compound-statement*

if-guard:
`if (` *condition* `)`

Rules are only ready to fire if the rule's guard is true and all the guards on methods invoked within the rule are also true.

```
void request.say(__int(32) v) if(!busy) {
    itemSay = v;
    ...
}
```

2.4 Connecting exported interfaces to imported references

The ___connect statement allows exported interface declarations to be connected with imported interface references between objects within a module declaration.

connect-declaration:
`__connect` *identifier* `=` *identifier* `;`

[*Example*:

AtomicC example

```
__interface ExampleRequest {
    void say(__int(32) v);
};

__module A {
     ExampleRequest callIn;
};

__module B {
    ExampleRequest *callOut;
};

__module C {
    A consumer;
    B producer;
    __connect producer.callOut = consumer.callIn;
};
```

BSV example

```
BSV example
BSV example
BSV example
BSV example
BSV example
```

— *end example*]

Comparision with BSV:

— The declaration for 'A' is just like BSV. In BSV, the declaration for B requires the interface instance for 'callOut' be passed in as an interface parameter (forcing a textual ordering to the source code declaration sequence).

— In AtomicC, the interfaces are stitched together outside in any convenient sequence in a location where both the concrete instances for A and B are visible.

2.5 Exporting interfaces from contained objects

In a design, there are times when the engineer wishes to declare an object locally, but allow external modules to access specific interfaces of the local object. This is done by declaring an interface to the containing object of compatible type and just 'assigning' the local object's interface to it.

[*Example*:

```
__module CWrapper {
    A consumer;
    ExampleRequest request = A.callIn;
 };
```

— *end example*]

CWrapper just forwards the interface 'request' down into the instance 'consumer'.

2.6 Syntax extension to C++

atomicc-class-key:

```
__interface
__emodule
__module
```

2.7 Exporting interfaces for use by software

In systems that have both hardware and software components, there is a need to marshall/demarshall parameterized method invocations across a hardware bus or network-on-chip (NOC). AtomicC provides this with my decorating the interface declarations with the keyword "___software".

The use of the ___software keyword causes the following to be performed:

— The generation of serialization/deserialization code for both software and hardware side modules to allow the method invocations to be performed in each direction

— The generation of header files allowing compilation of software modules that interface with the hardware

— Integration into a modified Connectal execution framework for the orchestration of requests.

[*Example*:

```
__module Echo {
    __software EchoRequest      request;              // exported interface
    __software EchoIndication   *indication;          // imported interface
    bool busy;
    __int(32) itemSay;
```

```
        ...
        // implementation of method request.say(). Note the guard "if (!busy)".
        void request.say(__int(32) v) if(!busy) {
            itemSay = v;
            ...
        }
        void request.saw(__int(16) a, __int(16) b) if(!busy) {
            ...
        }
    };
```

— *end example*]

[*Example*:

```
#include "EchoIndication.h"  // Header file generated by AtomicC
#include "EchoRequest.h"     // Header file generated by AtomicC

class EchoIndication : public EchoIndicationWrapper
{
public:
    virtual void heard(uint32_t v) {
        // user code for handling indication
    }
    EchoIndication(unsigned int id, PortalTransportFunctions *item, void *param) :
        EchoIndicationWrapper(id, item, param) {}
};

int main(int argc, const char **argv)
{
    EchoIndication echoIndication(IfcNames_EchoIndicationH2S, &transportMux, &param);
    EchoRequestProxy echoRequestProxy(IfcNames_EchoRequestS2H, &transportMux, &param);

    // user code for sending requests
    echoRequestProxy->say(42);
}
```

— *end example*]

3 Statements

3.1 ___rule

Rules specify a group of operations that must execute as an atomicly. A rule operates transactionally: when a rule's guard and the guards of all of its method invocations are satisfied, then it is ready to fire. It will fire on a clock cycle when it does not conflict with any higher priority rule.

rule-statement:
`__rule` *identifier* *if-guard*$_{opt}$ *compound-statement*

[*Example*:

```
__rule respond_rule if (responseAvail) {
    fifo->out.deq();
    ind->heard(fifo->out.first());
}
```

— *end example*]

3.2 Restrictions on C++ statements

Unlike the serialized execution model of C++, AtomicC supports a fully parallel, single cycle execution of rules which satisfy which are able to fire.

Since AtomicC does not generate any extra logic to support sequential execution behavior from language constructs, traditional C++ statements with non-static control flow behavior are not supported.

Examples include:

- Non-constant bound "for" statements. Constant bound "for" statements that can be fully unrolled are supported.
- "do", "while" statements
- Usages of "goto" that result in a cyclic directed graph of execution blocks
- Method and function calls that are not inlinable at compilation time (for example, recursion is prohibited)

4 Modularization

4.1 Independant compilation of modules

The design is separated into modules that can export and import interfaces to other modules. Each source language module compiles into a single verilog module. Modules are independantly compiled, depending only on the interface definitions for referenced modules. Referencing modules do not depend on the internal implementation of referenced modules, even if they textually exist in the same compilation unit. Scheduling of rules in a module is performed "inside out", with the resulting schedule dependancies written to a metadata file during compilation.

Exported interfaces can be used in several ways:

— invoked directly by the instantiator of the module,

— forwarded transparently, becoming another exported interface of the instantiating module,

— 'connected' to an 'interface reference' of another module in the instantiating scope.

4.2 Execution control

There are 2 common styles for communication of execution control information for a method:

— Asymmetric (ready/enable signalling) A method/rule is invoked by asserting the "enable" signal. This signal can only be asserted if the "ready" signal was valid, allowing the called module to restrict permissible execution sequences.

— Symmetric (ready/valid signalling) Both caller/callee have "able to be executed" signals. Execution is deemed to take place in each cycle where both "ready" (from the callee) and "valid" (from the caller) are asserted.

Bluespec uses the Asymmetric signalling style, collecting all scheduling control into a central location for analysis/generation. AtomicC uses the Symmetric signalling style, giving modules local control over their allowable execution patterns. Conflicts between local schedules for modules when they are connected together are detected by the linker.

4.3 Linking of groups of modules

To verify that an instantiated group of modules has SC compliant execution characteristics, a linker is used to cross check information from the metadata files for each module.

4.4 Interfacing with verilog modules

To reference a module in verilog, fields can be declared in ___interface items.

[*Example*:

```
__interface CNCONNECTNET2 {
    __input  __int(1)         IN1;
    __input  __int(1)         IN2;
    __output __int(1)         OUT1;
    __output __int(1)         OUT2;
};
```

```
__emodule CONNECTNET2 {
    CNCONNECTNET2 _;
};
```

— *end example*]

This will allow references/instantiation of an externally defined verilog module CONNECTNET2 that has 2 'input' ports, IN1 and IN2, as well as 2 'output' ports, OUT1 and OUT2.

4.4.1 Parameterized modules

Verilog modules that have module instantiation parameters can also be declared/referenced.

[*Example*:

```
__interface Mmcme2MMCME2_ADV {
    __parameter const char *  BANDWIDTH;
    __parameter float         CLKFBOUT_MULT_F;
    __input  __uint(1)        CLKFBIN;
    __output __uint(1)        CLKFBOUT;
    __output __uint(1)        CLKFBOUTB;
};
__emodule MMCME2_ADV {
    Mmcme2MMCME2_ADV _;
};
```

— *end example*]

This example can be instantiated as:

[*Example*:

```
__module Test {
    ...
    MMCME2_ADV#(BANDWIDTH="WIDE",CLKFBOUT_MULT_F=1.0) mmcm;
    ...
    Test() {
       __rule initRule {
           mmcm._.CLKFBIN = mmcm._.CLKFBOUT;
       }
    }
}
```

— *end example*]

4.4.2 Reference syntax

atomicc-method-declaration:

attribute-specifier-seq$_{opt}$ *pin-type*$_{opt}$ *decl-specifier-seq*$_{opt}$ *member-declarator-list*$_{opt}$;

pin-type:

`__input`
`__output`
`__inout`
`__parameter`

[*Example*:

```
__interface <interfaceName> {
     __input __uint(1) executeMethod;
     __input __uint(16) methodArgument;
     __output __uint(1) methodReady;
}
```

— *end example*]

For '___parameter' items, supported datatypes include: "const char *", "float", "int".

Factoring of interfaces into sub interfaces is also supported.

4.4.3 Clock/reset ports

Note that if interface port pins are declared in a module interface declaration, then CLK and nRST are _not_ automatically declared/instantiated. (Since the user needs the flexibility to not require them when interfacing with legacy code).

Note that this also allows arbitrary signals (like the output of clock generators) to be passed to modules as CLK/nRST signals. (For Atomicc generated modules, please note that the default clock/reset signals for a module will always have these names)

4.4.4 Import tooling

There is a tool to automate the creation of AtomicC header files from verilog source files. [*Example*:

```
atomiccImport -o MMCME2_ADV.h -C MMCME2_ADV -P Mmcme2 zynq.lib
atomiccImport -o VMMCME2_ADV.h -C MMCME2_ADV -P Mmcme2 MMCME2_ADV.v
```

— *end example*]

Annex A
Intermediate Representation

A.1 Module Definitions

atomicc-module-definition:
    module-type module-name `{` *module-body-list*$_{opt}$ `}`

module-type:
    `MODULE`
    `EMODULE`
    `INTERFACE`
    `STRUCT`
    `SERIALIZE`

module-body:
    module-body-definition
    module-body-definition module-body

module-body-definition:
    field-definition
    param-definition
    method-definition
    interface-definition
    interface-connect-definition

field-definition:
    `FIELD` *field-option-list*$_{opt}$ *field-type field-name*

param-definition:
    `PARAMS` *field-name* `<` *param-value* `>`

field-option-list:
    field-option
    field-option field-option-list

field-option:
    `/Ptr`
    `/shared`
    `/Count` *numeric-constant*
    verilog-field-options

verilog-field-options:
    `/parameter`
    `/input`
    `/output`
    `/inout`

interface-definition:
    `INTERFACE` *interface-option-list*$_{opt}$ *interface-type interface-name*

interface-connect-definition:
`INTERFACECONNECT` *connect-option-list*$_{opt}$ *interface-name-l interface-name-r interface-type*

connect-option-list:
connect-option
connect-option connect-option-list

connect-option-list:
`/Forward`

A.2 Method Definitions

method-definition:
`METHOD` *method-name method-options* `{` *method-contents-list* `}`

method-options:
method-flag-list$_{opt}$ *method-param-list*$_{opt}$ *method-return*$_{opt}$ *method-guard*$_{opt}$

method-flag-list:
method-flag
method-flag method-flag-list

method-flag:
`(`*/Rule)*
`(`*/Action)*

param-list:
param-definition
param-definition `,` *param-list*

param-definition:
param-type param-name

method-return:
return-type `=` *expression*

method-guard:
`if` *boolean-expression*

method-contents-list:
method-contents
method-contents method-contents-list

method-contents:
alloca-definition
let-definition
store-definition
call-definition

alloca-definition:
`ALLOCA` *alloca-type alloca-name*

let-definition:
`LET` *let-type action-guard*$_{opt}$ `:` *let-target-name* `=` *source-expression*

store-definition:
`STORE` *action-guard*$_{opt}$ `:` *store-target-name* `=` *source-expression*

call-definition:
`CALL` *call-option*$_{opt}$ *action-guard*$_{opt}$ `:` *call-target-name call-param-list*$_{opt}$

action-guard:
(*boolean-expression*)

Annex B
Scheduling example

B.1 Source program

```
__interface UserRequest {
    void say(__uint(32) va);
};

__module Order {
    UserRequest                     request;
    __uint(1) running;
    __uint(32) a, outA, outB, offset;
    void request.say(__uint(32) va) if (!running) {
        a = va;
        offset = 1;
        running = 1;
    }
    __rule A if (!__valid(request.say)) {
        outA = a + offset;
        if (running)
            a = a + 1;
    };
    __rule B if (!__valid(request.say)) {
        outB = a + offset;
        if (!running)
            a = 1;
    };
    __rule C if (!__valid(request.say)) {
        offset = offset + 1;
    };
};
```

B.2 Constraint graph

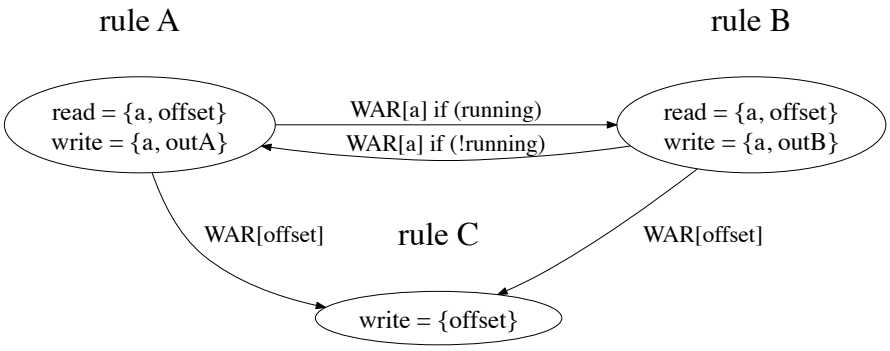

Figure 1 — Simple ordering example

Sequentially consistent schedules are:

— when 'running == 1': A -> B -> C
— when 'running == 0': B -> A -> C

B.3 Verilog output

```
module Order (input wire CLK, input wire nRST,
    input wire request$say__ENA,
    input wire [31:0]request$say$va,
    output wire request$say__RDY);
    reg [31:0]a, offset, outA, outB;
    reg running;
    assign request$say__RDY = !running;

    always @( posedge CLK) begin
      if (!nRST) begin
        a <= 0;
        offset <= 0;
        outA <= 0;
        outB <= 0;
        running <= 0;
      end // nRST
      else begin
        if (request$say__ENA == 0) begin // RULE$A___ENA
            outA <= a + offset;
            if (running != 0)
            a <= a + 1;
        end; // End of RULE$A___ENA
        if (request$say__ENA == 0) begin // RULE$B___ENA
            outB <= a + offset;
            if (running == 0)
            a <= 1;
        end; // End of RULE$B___ENA
        if (request$say__ENA == 0) begin // RULE$C___ENA
            offset <= offset + 1;
        end; // End of RULE$C___ENA
        if (request$say__ENA & ( !running )) begin // request$say___ENA
            a <= request$say$va;
            offset <= 1;
            running <= 1;
        end; // End of request$say___ENA
      end
    end // always @ (posedge CLK)
endmodule
```

B.4 Intermediate representation output

```
EMODULE l_ainterface_OC_UserRequest {
    METHOD/Action say__ENA ( INTEGER_32 va )
}
MODULE Order {
    INTERFACE l_ainterface_OC_UserRequest request
    FIELD INTEGER_1 running
    FIELD INTEGER_32 a
    FIELD INTEGER_32 outA
    FIELD INTEGER_32 outB
    FIELD INTEGER_32 offset
    METHOD/Rule/Action RULE$A__ENA if ((((request$say__ENA) != (0)) ^ (1))) {
        STORE :outA = (a) + (offset)
        STORE ((running) != (0)):a = (a) + (1)
    }
    METHOD/Rule/Action RULE$B__ENA if ((((request$say__ENA) != (0)) ^ (1))) {
        STORE :outB = (a) + (offset)
        STORE (((running) != (0)) ^ 1):a = 1
    }
    METHOD/Rule/Action RULE$C__ENA if ((((request$say__ENA) != (0)) ^ (1))) {
        STORE :offset = (offset) + (1)
    }
    METHOD/Action request$say__ENA ( INTEGER_32 va ) if ((((running) != (0)) ^ (1))) {
        STORE :a = request$say$va
        STORE :offset = 1
        STORE :running = 1
    }
}
```

Bibliography

[1] J. C. Hoe, "Operation-Centric Hardware Description and Synthesis," Ph.D. dissertation, MIT, Cambridge, MA, 2000.

[2] J. C. Hoe and Arvind, "Operation-Centric Hardware Description and Synthesis," *IEEE TRANSACTIONS on Computer-Aided Design of Integrated Circuits and Systems*, vol. 23, no. 9, September 2004.

[3] N. Dave, Arvind, and M. Pellauer, "Scheduling as rule composition," in *Proceedings of the 5th IEEE/ACM International Conference on Formal Methods and Models for Codesign*, ser. MEMOCODE '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 51–60.

[4] Bluespec Inc., `http://www.bluespec.com`.

[5] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Commun. ACM*, vol. 15, no. 12, pp. 1053–1058, Dec. 1972.

[6] M. C. Ng, K. E. Fleming, M. Vutukuru, S. Gross, Arvind, and H. Balakrishnan, "Airblue: A system for cross-layer wireless protocol development," in *Proceedings of the 6th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ser. ANCS '10. New York, NY, USA: ACM, 2010, pp. 4:1–4:11.

[7] M. Abbas and V. Betz, "Latency insensitive design styles for fpgas," in *28th International Conference on Field Programmable Logic and Applications, FPL 2018, Dublin, Ireland, August 27-31, 2018*, 2018, pp. 360–367.

[8] R. S. Nikhil, "Formal specification of bsv's elaboration and dynamic semantics," https://github.com/rsnikhil/Bluespec_BSV_Formal_Semantics, 2015.

[9] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[10] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Trans. Comput.*, vol. 28, no. 9, pp. 690–691, Sep. 1979.

[11] B. Liskov and S. Zilles, "Programming with abstract data types," in *SIGPLAN Notices*, 1974, pp. 50–59.

[12] N. Dave, "Designing a Reorder Buffer in Bluespec," in *Proceedings of MEMOCODE'04*, San Diego, CA, 2004.

[13] B. Alpern, M. N. Wegman, and F. K. Zadeck, "Detecting equality of variables in programs," in *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '88. New York, NY, USA: ACM, 1988, pp. 1–11.

[14] A. W. Appel, "Ssa is functional programming," *ACM SIGPLAN NOTICES*, vol. 33, no. 4, pp. 17–20, 1998.

[15] A. Prinz and B. Thalheim, "Operational semantics of transactions," in *IN CRPITS'17: PROCEEDINGS OF THE FOURTEENTH AUSTRALASIAN DATABASE CONFERENCE ON DATABASE TECHNOLOGIES 2003*, 2003, pp. 169–179.

[16] M. King, J. Hicks, and J. Ankcorn, "Software-driven hardware development," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2015, pp. 13–22.

[17] C. Fletcher, "Eecs150: Interfaces: "fifo" (a.k.a. ready/valid)," https://inst.eecs.berkeley.edu/~cs150/Documents/Interfaces.pdf, 2009.

[18] L. ARM, "Amba axi and ace protocol specification," https://developer.arm.com/docs/ihi0022/d/amba-axi-and-ace-protocol-specification-axi3-axi4-and-axi4-lite-ace-and-ace-lite, 2011.

[19] M. T. Özsu and P. Valduriez, *Principles of Distributed Database Systems, Third Edition*. Springer, 2011.

[20] D. Rosenkrantz, R. Stearns, and P. Lewis II, "Consistency and serializability in concurrent database systems," *SIAM Journal on Computing*, vol. 13, no. 3, pp. 508–530, 1984.

[21] H. W. Cain, M. H. Lipasti, and R. Nair, "Constraint graph analysis of multithreaded programs," in *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 4–.

[22] T. Esposito, M. Lis, R. Nanavati, J. Stoy, and J. Schwartz, "System and method for scheduling TRS rules," United States Patent US 133051-0001, February 2005.

[23] D. L. Rosenband and Arvind, "Hardware Synthesis from Guarded Atomic Actions with Performance Specifications," in *Proceedings of ICCAD'05*, San Jose, CA, 2005.

[24] R. Milner, M. Tofte, and R. Harper, *The Definition of Standard ML*. Cambridge, MA, USA: MIT Press, 1990.

[25] J. Gillenwater, G. Malecha, C. Salama, A. Y. Zhu, W. Taha, J. Grundy, and J. O'Leary, "Synthesizable high level hardware descriptions: Using statically typed two-level languages to guarantee verilog synthesizability," in *Proceedings of the 2008 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, ser. PEPM '08. New York, NY, USA: ACM, 2008, pp. 41–50.

[26] K. Memarian, J. Matthiesen, J. Lingard, K. Nienhuis, D. Chisnall, R. N. M. Watson, and P. Sewell, "Into the depths of c: Elaborating the de facto standards," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '16. New York, NY, USA: ACM, 2016, pp. 1–15.

[27] F. Pfenning, "Lecture notes on static semantics," https://www.cs.cmu.edu/~fp/courses/15411-f14/lectures/12-static.pdf, 2014.

[28] Arvind, "Bluespec-2: Two-level compilation and scheduling of operations," http://csg.csail.mit.edu/IAPBlue/handouts/Blue2-Compilation.pdf, 2003.