# Modular Full-System Verification of Hardware

Muralidaran Vijayaraghavan and Joonwon Choi

## 1 Introduction

Verification of systems can mean different things depending on the properties being verified. On one hand, one can verify if a system satisfies certain safety properties (for example, a program never divides by zero) and on the other hand, we have full-system verification (for example, a sorting function sorts its inputs, *i.e.,* create an ordered permutation of the input). This paper focuses on the problem of full-system verification for hardware designs.

What does a full-system verification for a hardware design entail? Does it mean verifying an exhaustive list of properties that a design must obey? Another way of doing full-system verification involves describing two systems: a simpler *specification* and a more complex *implementation* and showing that the implementation is *indistinguishable* from the specification. Our methodology proposes using the latter technique for verification.

This brings to the question of how to give a hardware specification and its implementation, and what the respective semantics are. Our proposition is to use the same language to describe both the specification and implementation and prove a *simulation relation* from the implementation to the specification, *i.e.,* a relation between a state in the implementation to the state in the specification iff the state reached by the state transitions in the implementation is also reachable by the state transitions in the specification.

State transitions are a straightforward concept when it comes to sequential systems. However, hardware systems are highly parallel, so a discussion of the specification of state transitions in hardware systems is warranted.

The commonly used technique for describing hardware systems is in the form of Register-Transfer-Level (RTL) descriptions. Here, all the (finite) state present in the hardware system is specified, along with the inputs and outputs for the overall system, and a state transition is specified in terms of the current state, and the current set of inputs, producing a new state and a new set of outputs.

The above description requires global specifications. But realistic systems are too big to be designed using global specifications, and must instead be broken into smaller modules. These modules are again RTL descriptions, and they are interconnected by joining the inputs of one modules to outputs of another modules, making sure that there are no cycles comprising of inputs and outputs which does not begin or end with a state element (these are called combinational cycles).

While RTL-based languages (like Verilog and VHDL) still dominate the industry for specifying hardware, there is a severe shortcoming in designing hardware using

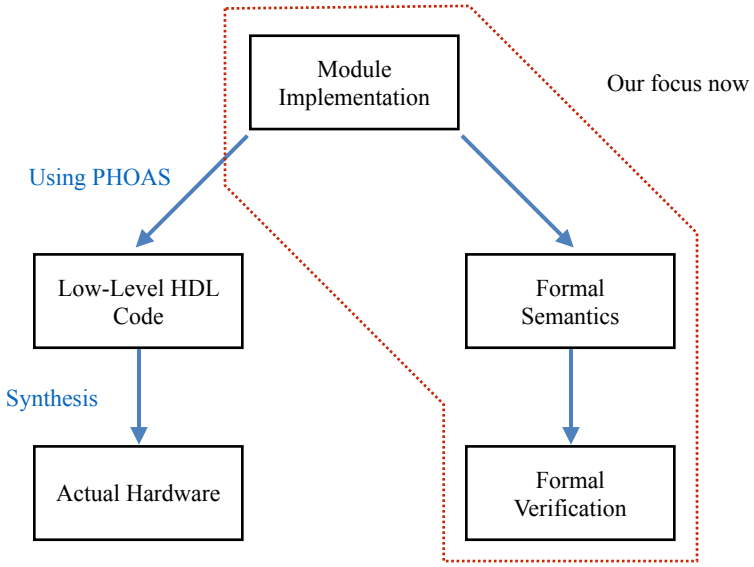


Figure 1: The overall roadmap of the project

these languages. RTL descriptions are synchronous in that they tie the functionality of a design with the timing behavior. For instance, if a register state is added to the output of say a multiplier module, the rest of the system that the multiplier is connected to will receive the output of the multiply operation in the next transition (i.e. clock cycle, in terms of hardware) and thus has to be redesigned, making the entire system fragile.

An alternative way to specify hardware is using guarded atomic actions à la Bluespec [1]. Here, the hardware system is specified in terms of several atomic transactions or *rules*, each of which describes how the state of the system changes when the rule fires, *i.e.,* the transaction happens. The state reached by the overall system is given by some sequence of firing of rules. The hardware circuit generated by a Bluespec compiler is smart enough to schedule non-conflicting transactions concurrently, thus improving performance.

We are going to use Bluespec-like descriptions as a starting point for hardware descriptions as these specifications are not as fragile as RTL descriptions. We developed a DSL in Coq for writing Bluespec-like specifications. We give modular semantics for this language, *i.e.,* semantics of a module without knowing about the other modules in the system. The semantics relate these modules to labeled transition systems (LTSes) [3, 4], a well known theory for describing communicating processes.

Our overall goal is to make this Bluespec-like DSL in Coq expressive enough to describe hardware systems easily. From these descriptions, we would like to generate hardware circuits (in Verilog) eventually. But in this project, we focused only on developing a modular semantics for this language and verifying simple hardware system examples using the language and its semantics. Figure 1 gives the overall

roadmap of this project and what we have achieved in the course of the 6.888 class. Before taking this class, we had done preliminary research to make all the decisions that we have taken for this class, namely using Bluespec-like descriptions, and the feasibility of this approach to verify complex systems. This preliminary research was submitted earlier to CAV, and has since been accepted. We are also attaching a copy of that paper for reference.

**Paper Organization** We start with a description of the syntax and semantics of the Bluespec-like DSL language in Section 2. We also describe the various issues in describing modular semantics for this language, and how these issues were solved. In Section 3, we describe the examples that we verified, namely that a single processor connected to an atomic memory and a register file implements sequential consistency. Finally, in 4, we describe how we want to take this project in the future, including verification of a more realistic and complex example involving coherent cache hierarchy and multiple processors, and developing a verified compiler à la CompCert [2] to synthesize the descriptions into hardware circuits.

# 2 Labeled Transition System Language for Specifying Hardware

We begin this chapter by giving the syntax for specifying hardware. A hardware *module* is similar to a notion of class in well-known language systems such as C++/Java. Each module has its own local state, internal state transition mechanism, and methods dealing with external requests. Once we define the module, the next step for specifying hardware is to develop the way of *communication* among modules. We adopted the notion of LTS to define such characteristics.

We give a syntax for modular hardware specifications in Section 2.1, relating it with the semantics of LTS. In Section 2.2, we present the LTS semantics. In Section 2.3, we present the definition of *trace refinement*, to determine if one LTS implements another.

## 2.1 Syntax for the LTS language

A module in LTS language consists of registers with initial values, *methods* and *rules* (see Figure 2 for a a block diagram of a module). Methods and rules consist of *actions*. An action is either a call of a method of a different module, passing an *expression* as an argument, a variable-to-expression binding, writing of an expression into a register, a conditional action, an assertion or returning an expression (which is relevant only when the action is in a method's definition).

The expressions in this language correspond to register reads, constants or operations on other expressions. Symbol $\vec{\cdot}$ represents a list of elements. Henceforth we will use $r$ and $c$ as a representative symbol for registers and constants, respectively.

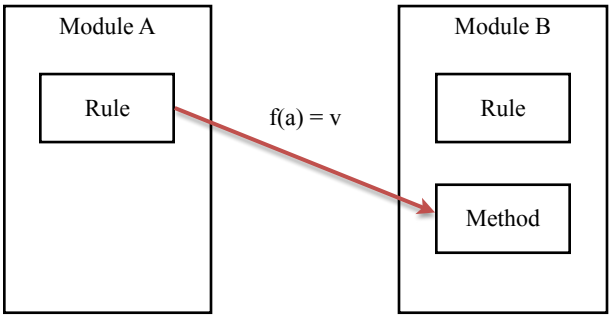


Figure 2: Communication of two modules

$$\begin{array}{lrcl} \text{Expression} & \langle e \rangle & ::= & r \\ & & | & c \\ & & | & op\ \vec{e} \end{array}$$

Actions comprise of register writes, method calls, expressing bindings, conditional actions, asserts and return actions. They are defined using the Continuation Passing Style (CPS) syntax. Each action except the return action contains a continuation which is terminated by the return action.

$$\begin{array}{lrcl} \text{Action} & \langle a \rangle & ::= & r := e;\ a \\ & & | & \textbf{let}\, x = f(e);\ a \\ & & | & \textbf{let}\, x = e;\ a \\ & & | & \textbf{if}\, e\, \textbf{then}\, a\, \textbf{else}\, a;\ a \\ & & | & \textbf{when}(e);\ a \\ & & | & \textbf{return}(e) \end{array}$$

Finally, a module is defines as follows:

$$\text{Basic Module} \quad \langle \mathcal{M} \rangle ::= \langle \overbrace{\langle r, c \rangle}^{\backslash};\ \overbrace{\langle g, a \rangle}^{\backslash};\ \overbrace{\langle f, \lambda x.a \rangle}^{\backslash} \rangle$$

Lastly, our entire hardware system consists of a tree hierarchy of modules, where the leaves are basic modules. Figure 2 actually shows two modules in the hierarchy, where the first module calls a method $f$ of the second module (the method calls are matched via names). The modules are composed using $\oplus$ and we can hide certain methods of a module using $m/\overrightarrow{f}$ which hides all modules other than those in the list $\overrightarrow{f}$. Overall, the composition is given by $m$ as follows:

$$\begin{array}{lrcl} & \langle m \rangle & ::= & \mathcal{M} \\ & & | & m \oplus m \\ & & | & m/\overrightarrow{f} \end{array}$$

The register names and method names are unique across all the modules which are hierarchically composed. For defining the semantics later, we want to be able to get the names of the registers and methods defined in a hierarchical module. We

also want to get the initial value for all registers in a hierarchical module. Registers can be obtained using function regs, rules obtained by rules, methods obtained by defs, and the initial values obtained by the regInits function. We use function $\pi_i$ to project the $i^{th}$ value in a tuple (starting with 1), and for a non-tuple single value, $\pi_1$ returns that value.

$$\begin{aligned}
\mathsf{regs}\ \mathcal{M} &= \mathsf{map}\ \pi_1\ (\mathsf{map}\ \pi_1\ \mathcal{M}) \\
\mathsf{regs}\ (m_1 \oplus m_2) &= \mathsf{regs}\ m_1 \cup \mathsf{regs}\ m_2 \\
\mathsf{regs}\ (m/p) &= \mathsf{regs}\ m
\end{aligned}$$

$$\begin{aligned}
\mathsf{rules}\ \mathcal{M} &= \mathsf{map}\ \pi_1\ (\mathsf{map}\ \pi_2\ \mathcal{M}) \\
\mathsf{rules}\ (m_1 \oplus m_2) &= \mathsf{defs}\ m_1 \cup \mathsf{defs}\ m_2 \\
\mathsf{rules}\ (m/p) &= \mathsf{defs}\ m
\end{aligned}$$

$$\begin{aligned}
\mathsf{defs}\ \mathcal{M} &= \mathsf{map}\ \pi_1\ (\mathsf{map}\ \pi_3\ \mathcal{M}) \\
\mathsf{defs}\ (m_1 \oplus m_2) &= \mathsf{defs}\ m_1 \cup \mathsf{defs}\ m_2 \\
\mathsf{defs}\ (m/p) &= \mathsf{defs}\ m \cap p
\end{aligned}$$

$$\begin{aligned}
\mathsf{regInits}\ \mathcal{M} &= \pi_1\ \mathcal{M} \\
\mathsf{regInits}\ (m_1 \oplus m_2) &= \mathsf{regInits}\ m_1 \cup \mathsf{regInits}\ m_2 \\
\mathsf{regInits}\ (m/p) &= \mathsf{regInits}\ m
\end{aligned}$$

## 2.2 Semantics of LTS

### 2.2.1 Semantics for Expression

We first give the deterministic, denotational semantics of evaluating an expression when given a map of register values. $[\![op]\!]$ returns a function which is semantically equivalent to the operation performed by $op$ on the list of expressions. It takes the current register mapping $\sigma$ as an argument.

$$\begin{aligned}
[\![r]\!]\ \sigma &= \sigma\ r \\
[\![c]\!]\ \sigma &= c \\
[\![op(es)]\!]\ \sigma &= [\![op]\!]\ (\mathsf{map}\ (\lambda e.([\![e]\!]\ \sigma))\ es)
\end{aligned}$$

### 2.2.2 Semantics for Action

Next we give the nondeterministic, relational semantics of an action $a$, which are given by the following judgments denoting transition relations from the current register mapping $o$ to a set of register updates $u$, with an indication of the methods from other modules (composed with the current module using $\oplus$) being called $cs$ along with the arguments passed and the return values expected and the return value of the action $v$. The judgment for actions are of the form: $o \vdash a \xrightarrow{cs} \langle u, v \rangle$. One point to note in a judgment for actions is that the return values of the called methods $cs$ will eventually be supplied by the execution of the method defined in the module being composed with the current module using $\oplus$; they are effectively unknown free-variables for the current action at this point.

We use $a \# b$ to denote $\mathsf{map}\ \pi_1\ a \cap \mathsf{map}\ \pi_1\ b = \{\}$, *i.e.,* given two sets of tuples (or single values) $a$ and $b$, their first values (or the only values) are disjoint.

$$\text{ActionWriteReg}\ \frac{o \vdash a \xrightarrow{cs} \langle u, v\rangle \qquad \langle r, \_\rangle \notin u}{o \vdash r := e;\ a \xrightarrow{cs} \langle \langle r, [\![e]\!]\ o\rangle :: u, v\rangle}$$

$$\text{ActionAssert}\ \frac{[\![e]\!]\ o \qquad o \vdash a \xrightarrow{cs} \langle u, v\rangle}{o \vdash \mathbf{when}(e);\ a \xrightarrow{cs} \langle u, v\rangle}$$

$$\text{ActionReturn}\ \frac{}{o \vdash \mathbf{return}(e) \xrightarrow{\{\}} \langle o, [\![e]\!]\ o\rangle}$$

$$\text{ActionBind}\ \frac{o \vdash a[([\![e]\!]\ o)/x] \xrightarrow{cs} \langle u, v\rangle}{o \vdash \mathbf{let}\, x = e;\ a \xrightarrow{cs} \langle u, v\rangle}$$

$$\text{ActionTrue}\ \frac{\begin{array}{ccc} o \vdash a_T \xrightarrow{cs_T} \langle u_T, v_T\rangle & o \vdash a[v_T/x] \xrightarrow{cs} \langle u, v\rangle & \\ [\![e]\!]\ o & u_T \# u & cs_T \# cs \end{array}}{o \vdash \mathbf{let}\, x = \mathbf{if}\ e\ \mathbf{then}\ a_T\ \mathbf{else}\ a_F;\ a \xrightarrow{cs_T \cup cs} \langle u_T \mathbin{+\!\!+} u, v\rangle}$$

$$\text{ActionFalse}\ \frac{\begin{array}{ccc} o \vdash a_F \xrightarrow{cs_F} \langle u_F, v_F\rangle & o \vdash a[v_F/x] \xrightarrow{cs} \langle u, v\rangle & \\ \neg([\![e]\!]\ o) & u_F \# u & cs_F \# cs \end{array}}{o \vdash \mathbf{let}\, x = \mathbf{if}\ e\ \mathbf{then}\ a_T\ \mathbf{else}\ a_F;\ a \xrightarrow{cs_F \cup cs} \langle u_F \mathbin{+\!\!+} u, v\rangle}$$

$$\text{ActionCall}\ \frac{o \vdash a[v'/x] \xrightarrow{cs} \langle u, v\rangle \qquad \langle f, \_, \_\rangle \notin cs}{o \vdash \mathbf{let}\, x = f(e);\ a \xrightarrow{cs \cup \{\langle f, [\![e]\!]\ o, v'\rangle\}} \langle u, v\rangle}$$

### 2.2.3 Semantics for Basic Module

From this, we define the semantics for a module. The judgments define a transition from current register mapping $o$ to a set of register updates $u$, with an indication of the defined method along with the argument expected and the returned value, the methods from other modules being called $cs$ along with the arguments passed and the return values expected, and whether the current transition is happening because of the rule $g$ or because of a method (indicated by $\mathbf{Rule}(g)$ or $\mathbf{Meth}$). We also allow combining multiple methods; this is needed to permit calling of multiple methods of a module from another module's rule or method, the latter module composed with the previous module using $\oplus$. The judgments are of the form: $o \vdash \mathcal{M} \xrightarrow{\langle \ell, ds, cs\rangle} u$, where $\ell$ is either $\mathbf{Rule}(g)$ or $\mathbf{Meth}$, and $ds$ is the set of methods whose transitions are allowed in this judgment. Just like how action judgments had a free-variable denoting the return values of the called methods, the judgment for a module has free-variables denoting the arguments for the methods defined in the module (which will be supplied by another module composed with the current module using $\oplus$).

As mentioned before, no register name or method name is being duplicated in a module, and the domain of the register mapping is given by the register names defined in the module. We will not litter the semantics by writing these conditions.

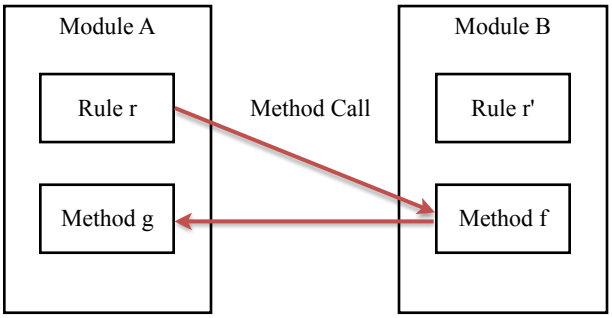


Figure 3: Communication of two modules with "call chains"

$$\text{Empty}\ \frac{}{o \vdash \mathcal{M} \xrightarrow{\langle \mathbf{Meth}, \{\}, \{\} \rangle} []}$$

$$\text{AddMeth}\ \frac{o \vdash \mathcal{M} \xrightarrow{\langle \mathbf{Meth}, ds, cs_1 \rangle} u_1 \quad \langle f, \lambda x.a \rangle \in \pi_3\, \mathcal{M} \quad o \vdash a[y/x] \xrightarrow{cs_2} \langle u_2, v \rangle}{u_1 \# u_2 \qquad f \notin ds \qquad cs_1 \# cs_2}}{o \vdash \mathcal{M} \xrightarrow{\langle \mathbf{Meth}, ds \cup \{f\}, cs_1 \cup cs_2 \rangle} u_1 \mathbin{++} u_2}$$

$$\text{AddRule}\ \frac{o \vdash \mathcal{M} \xrightarrow{\langle \mathbf{Meth}, ds, cs_1 \rangle} u_1 \quad \langle g, a \rangle \in \pi_2\, \mathcal{M} \quad o \vdash a \xrightarrow{cs_2} \langle u_2, \_ \rangle}{u_1 \# u_2 \qquad cs_1 \# cs_2}}{o \vdash \mathcal{M} \xrightarrow{\langle \mathbf{Rule}(g), ds, cs_1 \cup cs_2 \rangle} u_1 \mathbin{++} u_2}$$

### 2.2.4 Communication of modules with Labels

Once we are equipped with the semantics for basic modules, now we have to consider how to give semantics for communication among modules. As mentioned in Section 2.1, modules are combined like a binary tree, so inductively it is sufficient to deal with the case where two modules are combined.

Figure 2 shows the basic communication case between two modules. The rule in module $A$ calls the method $f$ with an argument value $a$ in module $B$, which returns value $v$. A function name, an argument, and the return value is sufficient ingredients to form a label, since each module has its own transition step independent to each other. In other words, state transitions (register map transition) are independently performed.

Nondeterministic and relational semantics for basic modules allow such communication with labels. For module $A$, it does not have any information on the return value so nondeterministically it assumes that value $v$ is returned. Conversely, for module $B$, it does not know what argument would be assigned so it also assumes that an argument $a$ is assigned. When combining two modules, we semantically check that two transition steps and a label match correctly. See 2.2.5 for formal definitions for such semantics.

Notice that in our semantics, the label associated with a transition of a module can contain multiple defined methods. Why do we need such a definition for a transition of a module? Consider a module composition as shown in Figure 3 where rule $r$ in module $A$ calls method $f$ in module $B$, and $f$ also calls method $g$ in $A$ once more. For module $A$, we need to reflect two transition steps: one for rule $r$, and the other for method $g$, because when $r$ calls method $f$ of module $B$, then it in turn calls $g$ of module $A$. This is made possible by having a combined transition for the rule $r$ and method $g$. This combined transition will combine with a transition denoting method $f$ in module $B$. We will discuss how to give a combined transition consisting of transitions of multiple methods and rules next.

### 2.2.5 Semantics for a Hierarchy of Modules

We thus define the semantics of a hierarchy of modules, which combines several transitions of methods and rules into one transition. For each constructor of a module hierarchy, we provide the semantics via judgments in the form of labeled transitions. The labels of the transition is given by $\langle \ell, ds, cs \rangle$, where $ds$ denotes the methods defined by the module that are taking part in the transition and $cs$ denotes the methods of other modules being called in this transition. Both $ds$ and $cs$ have the arguments and return values of each of the methods in them. We call a label empty when it is $\langle \_, \{\}, \{\} \rangle$ and denote it by $\epsilon$.

The judgment for the semantics of a hierarchy of modules is given by $o \xrightarrow[m]{\langle \ell, ds, cs \rangle} n$ where $m$ is the hierarchical module undergoing the transition, $o$ is the old state mapping, $n$ the new state mapping, $\ell$ is either $\mathbf{Rule}(g)$ or $\mathbf{Meth}$, $ds$ the defined methods in this transition, and $cs$ the called methods in this transition. The function $\mathsf{apply}\ o\ u$ applies the register updates $u$ on the register mapping $o$, as follows:

$$\begin{array}{rcl} \mathsf{apply}\ o\ [] & = & o \\ \mathsf{apply}\ o\ (\langle r, v \rangle :: u) & = & \mathsf{apply}\ (o[r := v])\ u \end{array}$$

There are three judgments for the semantics for hierarchy of modules. The first one is for a single basic module; we get the semantic by naturally lifting the semantics for basic modules (SingleModule). A semantic for method-hidden module is also defined easily by checking whether each method in $ds$ belongs to the filter (HideModule).

$$\text{SingleModule}\ \frac{o \vdash \mathcal{M} \xrightarrow{\langle \ell, ds, cs \rangle} u}{o \xrightarrow[\mathcal{M}]{\langle \ell, ds, cs \rangle} \mathsf{apply}\ o\ u}$$

$$\text{HideModule}\ \frac{o \xrightarrow[m]{\langle \ell, ds, cs \rangle} n \qquad \langle f, \_, \_ \rangle \in ds \Rightarrow f \in h}{o \xrightarrow[m/h]{\langle \ell, ds, cs \rangle} n}$$

Finally, we give an judgment for combined modules, where both two modules undergo transitions. The rule or method of one module calls the methods of another module, which in turn can call methods of the first module. Note that the only

judgment for combined modules includes the case where one module progresses while another one remains the same, since it is exactly the "EmptyMeth" case.

We define a predicate called defCallEq to denote that the argument and return value of a called method in a judgment used for module composition is exactly equal to the defined method's argument and return value, respectively. This is useful in defining the judgment for combined modules.

$$\begin{aligned}\mathsf{defCallEq}\ fs\ ds\ cs \coloneqq \forall f \in fs.\ \langle f, \_, \_\rangle \in cs \Rightarrow \\ \langle f, \_, \_\rangle \in ds \wedge \forall a. \forall v.\ \langle f, a, v\rangle \in ds \Leftrightarrow \langle f, a, v\rangle \in cs\end{aligned}$$

The function rmMeths $x$ $y$ removes the methods of $x$ that also occur in $y$. The sets $x$ and $y$ are 3-tuples $\langle f, a, v\rangle$ where $f$ denotes the method-name, $a$ the argument and $v$ the return value. rmMeths matches for common method names; the arguments are return values can be different.

$$\mathsf{rmMeths}\ x\ y \coloneqq \{\langle f, a, v\rangle | \langle f, a, v\rangle \in x \wedge \langle f, \_, \_\rangle \notin y\}$$

We are now in a position to combine transitions from two different modules. Note that once a defined method is called by an action in another module, it can no longer be called by any other action taking part in the same overall transition; the defined method is effectively hidden.

$$\text{Combine}\ \frac{\begin{array}{c} o_1 \xrightarrow[m_1]{\langle \ell_1, ds_1, cs_1\rangle} n_1 \qquad o_2 \xrightarrow[m_2]{\langle \ell_2, ds_2, cs_2\rangle} n_2 \\ \neg(\exists g_1.\ \ell_1 = \mathbf{Rule}(g_1) \wedge \exists g_2.\ \ell_2 = \mathbf{Rule}(g_2)) \\ \mathsf{defCallEq}\ (\mathsf{defs}\ m_2)\ ds_2\ cs_1 \qquad \mathsf{defCallEq}\ (\mathsf{defs}\ m_1)\ ds_1\ cs_2 \end{array}}{o_1 \uplus o_2 \xrightarrow[m_1 \oplus m_2]{\langle \ell, ds, cs\rangle} n_1 \uplus n_2}$$

where
$$\begin{aligned} \ell &= \mathbf{if}\ \ell_1 = \mathbf{Rule}(\_)\ \mathbf{then}\ \ell_1\ \mathbf{else\ if}\ \ell_2 = \mathbf{Rule}(\_)\ \mathbf{then}\ \ell_2\ \mathbf{else\ Meth} \\ cs &= \mathsf{rmMeths}\ cs_1\ ds_2 \cup \mathsf{rmMeths}\ cs_2\ ds_1 \\ ds &= \mathsf{rmMeths}\ ds_1\ cs_2 \cup \mathsf{rmMeths}\ ds_2\ cs_1 \end{aligned}$$

### 2.2.6 Transition Closure

From a judgment representing single-step module evolution, we can build a judgment capturing arbitrary-length evolutions.

**Definition 1. Transitive-Reflexive Closure**: If module $m$ steps from state $o$ to a state $n$ using zero or more transitions, then we say that $o \xrightarrow[m]{\sigma} {}^* n$, where the label $\sigma$ is a *sequence* of labels given by the concatenation of all non-empty labels generated in $m$ during the course of the transitions.

$$^*\text{Nil}\ \frac{}{o \xrightarrow[m]{[]} {}^* o}$$

$$\text{*Empty}\ \frac{o \xrightarrow[m]{\sigma}{}^* n \qquad n \xrightarrow[m]{\epsilon} n'}{o \xrightarrow[m]{\sigma}{}^* n'}$$

$$\text{*NonEmpty}\ \frac{o \xrightarrow[m]{\sigma}{}^* n \qquad n \xrightarrow[m]{\alpha} n' \qquad \alpha \neq \epsilon}{o \xrightarrow[m]{\alpha::\sigma}{}^* n'}$$

**Definition 2. Evolution** of $m$: If $m$ steps from its initial state to a state $n$ using zero or more transitions, then we say that $m \rightarrow^* \langle n, \sigma \rangle$, where the label $\sigma$ is a *sequence* of labels given by the concatenation of all non-empty labels generated in $m$ during the course of the transitions.

$$\text{Evolution}\ \frac{\lambda x.(\llbracket \mathsf{find}\ x\ (\mathsf{reglnits}\ m) \rrbracket\ \_) \xrightarrow[m]{\sigma}{}^* n}{m \rightarrow^* \langle n, \sigma \rangle}$$

## 2.3 Trace Refinement for LTS

We need a notion of when one LTS *implements* another. A system that produces identical labels as another under all circumstances, and has a mapping from its state to the other's state can be considered as safe substitutes for one another. We will define an asymmetrical notion of compatibility:

**Definition 3.** Let two LTSes $m_1$ and $m_2$ have the same label set $\mathcal{L} \subseteq \{\langle \ell, ds, cs \rangle\}$ (*i.e.,* same set of defined and called methods). Let $\rho : \mathcal{L} \rightarrow \mathcal{L}$ be a function that is able to replace labels with alternative labels, or erase them altogether (*i.e.,* make the label empty). We say that $m_1$ **trace-refines** $m_2$ **w.r.t.** $\rho$, or $m_1 \sqsubseteq_\rho m_2$, if:

$$\forall s_1. \forall \sigma.\ m_1 \rightarrow^* \langle s_1, \sigma \rangle \Rightarrow \exists s_2.\ m_2 \rightarrow^* \langle s_2, \hat{\rho}\ \sigma \rangle$$

All $\epsilon$-labels in $\rho$ is dropped before they are replaced by the mapping of $\rho$ on it, and labels mapped to $\epsilon$ by $\rho$ are also dropped. $\hat{\rho}$ is the overloaded version of $\rho$ for a sequence of labels when applied to $\sigma$.

As a shorthand, we write $m_1 \sqsubseteq m_2$ for $m_1 \sqsubseteq_{\mathsf{id}} m_2$, for $\mathsf{id}$ an identity function, forcing traces in the two systems to match exactly. Under this notion of identical traces, we say that $m_1$ is sound w.r.t. $m_2$.

### 2.3.1 Lemmas for Easy Trace Refinement Proofs

The trace refinement definition presented in the previous section is general, but is practically hard to use. Often, every transition step that an implementation makes can be made to map to a transition step of the specification. In this case, we can have a weaker theorem for trace refinement.

**Lemma 1.** *If there is a mapping $f$ from any state of the implementation $A$ to a state of the specification $B$, and for every transition step that the implementation takes, going from state $s_A$ to $s'_A$ producing label $\ell$, there is a transition step in the specification from state $(f\ s_A)$ to state $(f\ s'_A)$ producing label $(g\ \ell)$, then $A \sqsubseteq_g B$.*

The above theorem is still not very useful. According to this theorem, we have to prove the existence of mapped labels for all possible labels that the implementation can produce. More specifically, since labels can be produced by combined transition, we need to provide a mapping for all possible legal combinations of individual transitions. For example, if a module has two rules and three methods, we should give a mapping for each combined transition formed from the 3 methods with any one rule, or no rule, leading to a total of 24 combined transitions.

However, if there is a mapping from each rule and each method of the implementation to a rule or a method, respectively, of the specification, then we do not have to give a mapping for each combined transition; it would suffice to give a mapping for transitions comprising just individual rules or methods. However, in the implementation, a rule (or a method) can call methods of other modules – only the semantics obtained by inlining the called methods can match the semantics of the rule (or method) of a specification. These conditions are summarized in the following theorem:

**Lemma 2.** *If there is a mapping $f$ from any state of the implementation $A$ to a state of the specification $B$, and*

1. *for every "simple" transition step that the implementation takes, going from state $s_A$ to $s'_A$ producing label $\langle \textbf{\textit{Rule}}(x), \{\}, cs\rangle$, there is a rule $y$ in the specification from state $(f\ s_A)$ to state $(f\ s'_A)$ producing label $\langle \textbf{\textit{Rule}}(y), \{\}, cs\rangle$, and*

2. *for every "simple" transition step that the implementation takes, going from state $s_A$ to $s'_A$ producing label $\langle \textbf{\textit{Meth}}, \{\langle h, a, v\rangle\}, cs\rangle$, method $h$ in the specification also takes the state state $(f\ s_A)$ to $(f\ s'_A)$ producing label $\langle \textbf{\textit{Meth}}, \langle h, a, v\rangle, cs\rangle$,*

*then $A \sqsubseteq B$.*

# 3 Examples for the LTS Language

In this chapter we give two examples which employ the LTS language. We first present a simple one-element FIFO in Section 3.1. With this example we will have a better understanding for evaluating an action. Then we present a decoupled modular processor in Section 3.2. The processor consists of three modules - the main core, register file, and data memory. We will show that the composed processor is an implementation of sequential consistency.

## 3.1 An One-Element FIFO

FIFO (First-In-First-Out) is a simple structure that allows elements to be saved temporarily and ensures them to be used in order. In this section, we present the module definition for an one-element FIFO. The FIFO has only one element, which indicates that an old element is overwritten when we request to push a new element.

FIFO can be easily defined by the LTS language. We will have a register which holds the only element. With the register we can define `enq` and `deq` which requests

to push a value and to pop the value, respectively. The following defines the FIFO module $\mathcal{F}$:

$$\begin{array}{rcl} \mathsf{regs}\ \mathcal{F} & = & \langle \mathsf{data}, 0\rangle :: [] \\ \mathsf{rules}\ \mathcal{F} & = & \{\} \\ \mathsf{defs}\ \mathcal{F} & = & \langle \mathsf{enq}, \mathsf{enqBody}\rangle :: \langle \mathsf{deq}, \mathsf{deqBody}\rangle :: [] \end{array}$$

$$\begin{array}{rcl} \mathsf{enqBody} & := & \lambda x.(\mathsf{data} := x;\ \mathbf{return}(0)) \\ \mathsf{deqBody} & := & \lambda\_.(\mathbf{return}(\mathsf{data})) \end{array}$$

We have the only one register data which holds the data. There are no rules in $\mathcal{F}$. Two methods enq and deq are defined so they push and pop the data respectively.

Now we can prove a simple property of $\mathcal{F}$ that when we enqueue an element and dequeue right away, we may get the same element value. The theorem is stated as follows:

**Theorem 1** (A property of $\mathcal{F}$)**.** $\forall o.\forall u.\forall v.\ o \vdash \mathsf{enqBody}\ v \xrightarrow{\{\}} \langle u, \_\rangle \Rightarrow \mathsf{apply}\ o\ u \vdash \mathsf{deqBody} \xrightarrow{\{\}} \langle \_, v\rangle$

The proof is straightforward by the inference rules for actions. We first derive $u = \langle \mathsf{data}, v\rangle :: \{\}$ with the inference tree for enqBody, and substitute $u$ to the inference tree for deqBody to derive that the return value equals $v$, as follows:

$$\dfrac{\dfrac{}{o \vdash \mathbf{return}(0) \xrightarrow{\{\}} \langle\{\}, 0\rangle}\ \text{ActionReturn}}{o \vdash \mathsf{data} := v; \mathbf{return}(0) \xrightarrow{\{\}} \langle\langle \mathsf{data}, v\ \rangle :: \{\}, 0\rangle}\ \text{ActionWriteReg}$$

Figure 4: Inference tree for enqBody

$$\dfrac{}{\mathsf{apply}\ o\ (\langle \mathsf{data}, v\ \rangle :: []) \vdash \mathbf{return}(\mathsf{data}) \xrightarrow{\{\}} \langle\{\}, v\rangle}\ \text{ActionReturn}$$

Figure 5: Inference tree for deqBody

### 3.2 A Decoupled Modular Processor and Sequential Consistency

In this section we first modularly define components for a simple processor. As seen in Figure 6, the target processor comprises of the execution core, register file, instruction memory, and data memory. Finally we build the final processor by composing three modules: core, register file, and data memory.

Next we give a proof that the composed processor implements sequential consistency, by defining *a specification module* which defines sequential consistency and proving the trace refinement between them.

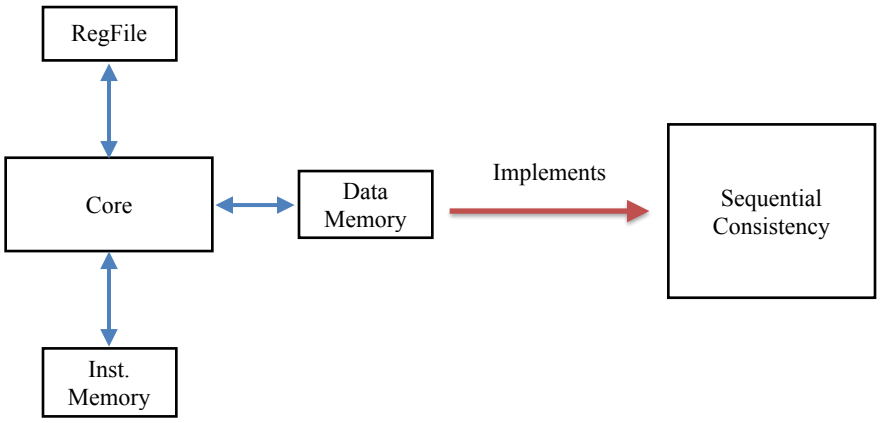


Figure 6: A decoupled processor and Sequential Consistency

### 3.2.1 A Decoupled Modular Processor

We introduce a number of simplifications which makes the proof easier:

- We do not have an instruction memory as a module; instead, we assume that there is a decode function dec parameter, which takes the Program Counter (PC) address as an argument and returns the instruction. In Coq, dec is passed as a Gallina term.
- We only have three kinds of instructions - load, store, and halt; this captures all the instructions relevant to Sequential Consistency.
- Lastly, we have no buffers (FIFOs) between any two components, which implies that there are no cycle delays for the register file or the memory. It simplifies the refinement proof since we do not have to introduce empty steps ($\epsilon$-labeled) to match progress steps.

**Core** The main core has a register pc which acts as PC, and a rule procExec for executing an instruction with respect to pc. procExec first decodes the instruction (dec pc), where dec is given as a parameter, and checks whether the instruction is load, store, or halt by seeing its opcode which is then handled accordingly. The formal definition for such a core $\mathcal{P}$ is given as follows:

$$\begin{aligned} \mathsf{regs}\ \mathcal{P} &= \mathsf{pc} :: [] \\ \mathsf{rules}\ \mathcal{P} &= (\mathsf{procExec}\ \mathsf{dec}) :: [] \\ \mathsf{defs}\ \mathcal{P} &= \{\} \end{aligned}$$

```
(procExec dec)  :=  λ_.  if opcodeOf (dec pc) = load
                         then let v = ReqLd(addrOf (dec pc));
                              let _ = RfWrite(regOf (dec pc), v);
                              return(0)
                         else if opcodeOf (dec pc) = store
                         then let _ = ReqSt(addrOf (dec pc), valOf (dec pc)); return(0)
                         else let _ = Halt(); return(0)
```

The core $\mathcal{P}$ may call a number of methods depending on the instruction. For instance, when (dec pc) is a load instruction, we first call ReqLd to bring the loaded value from the memory, and call RfWrite to save the value to the target register. All necessary information - address, register name, and store value - is included in (dec pc), where we can access their value by addrOf, regOf, and valOf.

**Register File and Memory** Register file and memory are implemented by a register storing a vector (which is a primitive expression in our LTS language). Register file has two methods RfRead and RfWrite for reading and writing register values, respectively. Memory also has two methods ReqLd and ReqSt for reading and writing values for addresses, respectively. Recall that all methods have no delays, *i.e.,* requests are handled and values are returned immediately.

### 3.2.2 A Specification Module: Sequential Consistency

In order to prove the processor implements Sequential Consistency (SC), we first have to formally define what it is. There are a number of ways to define SC, in this paper, we give the definition by implementing *a module* which will act as a specification for SC.

The module $\mathcal{SC}$ has three registers: a program counter, a register file and a memory. It has only one rule scExec, similar to the rule procExec in the core $\mathcal{P}$ in the previous section. scExec exactly does the same thing as procExec does. $\mathcal{SC}$ also gets dec as an argument to support the definition of scExec. It is defined formally as follows:

```
regs SC   =  pc :: rf :: mem :: []
rules SC  =  (scExec dec) :: []
defs SC   =  {}
```

```
(scExec dec)  :=  λ_.  if opcodeOf (dec pc) = load
                       then let v = mem (addrOf (decpc));
                            rf := rf[regOf (dec pc) := v];
                            return(0)
                       else if opcodeOf (dec pc) = store
                       then mem := mem[addrOf (dec pc) := valOf (dec pc)]; return(0)
                       else let _ = Halt(); return(0)
```

Notice that $\mathcal{SC}$ is basically a syntactially inlined version of the processor. That is, we can build $\mathcal{SC}$ by composing all registers in modules in $\mathcal{P}$, inlining all methods bodies in each module.

### 3.2.3 Refinement Proof

Equipped with the definitions for the processor and $\mathcal{SC}$, our goal is to prove that the processor $\sqsubseteq \mathcal{SC}$.

We use Theorem 2 for the proof. A register map should be defined to use this theorem; in this case, the map is like an identity function. This is because the registers in $\mathcal{SC}$ are exactly same as the composed ones in each module in the processor.

The only proof burden is to prove two itemized conditions in Theorem 2. However, it can be proven easily in this case since the two rules procExec and scExec have a similar structure; scExec is a syntactically inlined version of procExec, as mentioned in Section 3.2.2. And, the action in each of the methods of the processor maps exactly to those in $\mathcal{SC}$.

## 4 Future Works

There are two areas to expand this work: one is to develop lemmas and LTac tactics for easy verification of LTS in Coq, and the other is to verify the real-world hardware systems using this theory.

In spite of having designed and proved several lemmas, and developed several custom proof tactics for automation, we are still seeing a lot of proof processes should be abstracted and automated. This proof engineering will be performed during the verification for complex hardware systems. Our development is currently hosted in http://github.mit.edu/joonwonc/LtsLanguage.

Currently we aim to verify the sequential consistency of complex out-of-order processor connected to a multi-level coherent cache hierarchy. We expect that with our current modular syntax and semantics, verifying out-of-order processor and cache coherence protocol can be done *modularly*, which makes the whole system proof tractable. The CAV paper we attached describes the actual steps for verifying such a complex processor.

## References


[1] James C. Hoe and Arvind. Synthesis of operation-centric hardware descriptions. In *Proceedings of the 2000 IEEE/ACM International Conference on Computer-aided Design*, ICCAD '00, pages 511–519, Piscataway, NJ, USA, 2000. IEEE Press.

[2] Xavier Leroy and Sandrine Blazy. Formal verification of a C-like memory model and its uses for verifying program transformations. *Journal of Automated Reasoning*, 41(1):1–31, 2008.

[3] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980.

[4] Robin Milner. *Communication and concurrency*. PHI Series in computer science. Prentice Hall, 1989.