Assume-Guarantee Refinement Between Different Time Scales *

Thomas A. Henzinger ** Shaz Qadeer *** Sriram K. Rajamani †

{tah,shaz,sriramr}@eecs.berkeley.edu

Abstract. Refinement checking is used to verify implementations against more abstract specifications. Assume-guarantee reasoning is used to decompose refinement proofs in order to avoid state-space explosion. In previous approaches, specifications are forced to operate on the same time scale as the implementation. This may lead to unnatural specifications and inefficiencies in verification. We introduce a novel methodology for decomposing refinement proofs of temporally abstract specifications, which specify implementation requirements only at certain sampling instances in time. Our new assume-guarantee rule allows separate refinement maps for specifying functionality and timing. We present the theory for the correctness of our methodology, and illustrate it using a simple example. Support for sampling and the generalized assume-guarantee rule have been implemented in the model checker MOCHA and successfully applied to verify the VGI multiprocessor dataflow chip with 6 million transistors.

1 Introduction

Formal verification is a systematic approach for detecting logical errors in designs. The design is first described in a language with a mathematical semantics and then analyzed for correctness with respect to a specification. We refer to the design being analyzed as the implementation. The verification problem is called *refinement checking* when the specification is a more abstract design. The refinement-checking problem is PSPACE-hard in the size of the implementation description. Not surprisingly, algorithms for refinement checking are exponential in the size of the implementation description. This is the so-called *state-explosion* problem in verification.

Specifications are typically less detailed than the implementation. For example, the specification of an adder might simply state that the output is the sum of the two inputs, whereas the implementation might be a gate-level adder circuit, which operates at the detail of individual bits. Nonetheless, common notions of correctness require specifications to operate in "lock-step" with the

* This research was supported in part by the DARPA/NASA grant NAG2-1214, by the SRC contract 97-DC-324.041, by the NSF CAREER award CCR-9501708, by the ARO MURI grant DAAH-04-96-1-0341, and by an SRC fellowship of the third author.

** EECS Department, University of California, Berkeley, USA, and Max-Planck Institute for Computer Science, Saarbrücken, Germany

*** EECS Department, University of California, Berkeley, USA

† EECS Department, University of California, Berkeley, USA

implementation: every possible computation step of the implementation must be matched by an admissible computation step of the specification. If the natural time scale of the specification is less detailed than that of the implementation —for example, if the gate-level adder requires several clock cycles to compute a sum— then the specification often is "slowed down" by stuttering, even for perfectly synchronous designs. A prominent example of this occurs in pipeline verification, where the Instruction Set Architecture (ISA) specification usually is slowed down by introducing a nondeterministic *stall* signal to stretch its time scale to match that of the pipeline [HQR98,McM98]. Instead of slowing down the specification, we pursue the alternative of "speeding up" the implementation. For this purpose, we use an operator called *Sample*, which samples the behavior of the implementation at appropriately defined sampling instants.[1]

Our motivation for sampling arose specifically from the attempt of verifying a 96-processor V(ideo) G(raphics) I(mage) chip designed by the Infopad project at the University of California, Berkeley [STUR98]. There, the specification consists of ISAs for the individual processors and FIFO buffers that abstract the point-to-point communication protocols, which interact subtly with the processor pipelines. Since the implementation contains level-sensitive latches and different parts of the circuit are active at high vs. low phases of the clock, sampling must be used to match the implementation time scale with the specification time scale. While the computational advantages of sampling in state-space exploration have been demonstrated in [AHR98], the VGI is still far beyond the scope of exhaustive search. Hence, we needed to generalize a compositional verification methodology to accommodate the *Sample* operator.

Scalable approaches to refinement checking make use of the compositional structure of both implementation and specification, and divide the verification task at hand into simpler subtasks. Suppose we have a refinement-checking problem of the form $P_1 \| P_2 \preceq Q_1 \| Q_2$, and the state space of the implementation $P_1 \| P_2$, even when sampled, is too large to be handled by exhaustive search algorithms. A naive compositional approach would attempt to prove both $P_1 \preceq Q_1$ and $P_2 \preceq Q_2$, and then conclude that $P_1 \| P_2 \preceq Q_1 \| Q_2$. Though sound, the naive approach often fails in practice, because P_1 usually refines Q_1 only in a suitable constraining environment, and so does P_2. The constraining environments are taken into account by the *assume-guarantee* approach, which concludes $P_1 \| P_2 \preceq Q_1 \| Q_2$ from the two proof obligations $P_1 \| Q_2 \preceq Q_1$ and $Q_1 \| P_2 \preceq Q_2$ (the apparent circularity in such proofs is resolved by an induction over time) [Sta85,Kur94,AL95,AH96,McM97,HQR98]. Note that the assume-guarantee approach avoids constructing $P_1 \| P_2$. With assume-guarantee reasoning, the difficulty in refinement checking shifts from the sizes of the involved state spaces to the "semantic gap" between implementation and specification. To bridge this gap, one writes witness modules that map implementation sig-

[1] The *Sample* operator is similar, but not identical, to the *Next* operator of Reactive Modules [AH96]: while *Next* changes the time scale of a module *and* its environment, *Sample* does not constrain the environment, which therefore may offer multiple inputs between sampling instances.

nals to specification signals, and refinement constraints that relate specification signals to implementation signals [HQR98].

In this paper, we generalize the assume-guarantee method to accommodate the sampling operator. If implementation and specification operate at the same time scale, witness modules generate values for hidden specification variables at each step. However, if a single macro-step of the specification corresponds to several micro-steps of the implementation, it is necessary to provide witness modules that operate at the micro-step level. The purpose of such a witness is to generate the correct value for the specification signal to be witnessed and to maintain that value until the next sampling instance. Dually, if implementation and specification operate at the same time scale, refinement constraints provide abstract definitions for implementation variables at each step. If one specification step corresponds to several implementation steps, then it no longer suffices for the refinement constraints to supply values for the implementation variables at the rate of the specification —at sampling instances— but additional constraints need to be provided between sampling instances. Providing different refinement constraints at the macro and micro levels enables a separation of concerns: while macro-level constraints (at sampling instances) tend to describe the functional behavior of an implementation variable, micro-level constraints (between sampling instances) tend to describe its timing behavior. This separation of functionality and timing is particularly natural for collections of synchronous blocks that communicate asynchronously.

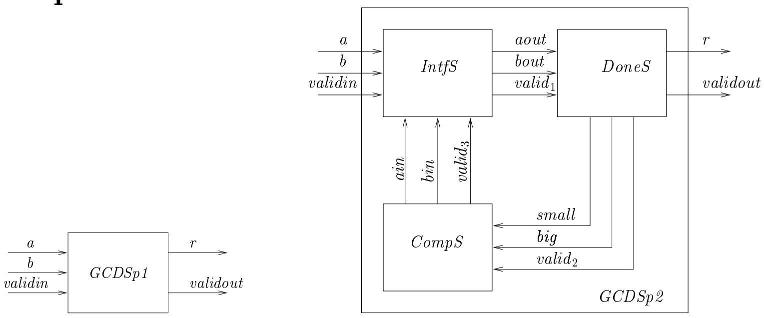
We develop the theory and methodology to carry out assume-guarantee reasoning when specifications are abstract in both space (fewer variables/components) and time (fewer observation points). The crux of the theory lies in the ability to distribute the *Sample* operator over the parallel composition of implementation components using micro-level refinement constraints. The resulting assume-guarantee proof rule produces refinement obligations both at the macro level and at the micro level, which are then discharged by our model checker MOCHA [AHM+98]. While micro-level refinement obligations can be handled using traditional model checking, we had to enhance MOCHA to discharge macro-level refinement obligations. We have used this methodology successfully in the verification of the VGI chip. We found several subtle bugs which were unknown to the designers. The case study describing this effort can be found in [HLQR99].

Speeding up the implementation using *Sample* has significant computational advantages when compared to earlier approaches that slow down the specification using stuttering. The sampled implementation typically has a smaller state space than the original implementation. If the refinement checking uses explicit state enumeration, we can use a secondary stack to explore and discard the implementation states between two sampling instances. If we use symbolic search using binary decision diagrams (BDDs), the sampled state space can be typically represented using a smaller BDD [AHR98].

Outline of the paper. Since the VGI is too complex for the purpose of exposing our methodology, we present a simple GCD example in Section 2. MOCHA operates in the heterogeneous modeling framework of Reactive Modules [AH96].

In order to introduce sampling, we have to generalize modules to so-called *transition constraints*, which need not be executable. This is done in Section 3. The *Sample* operator and its properties are introduced in Section 4. The assume-guarantee methodology of [McM97,HQR98,McM98] is generalized in Section 5 to accommodate the *Sample* operator, and the generalized methodology is applied in Section 6 to the GCD example. Finally, in Section 7, the VGI verification is described briefly.

2 Example



(a) High level specification

(b) Intermediate specification

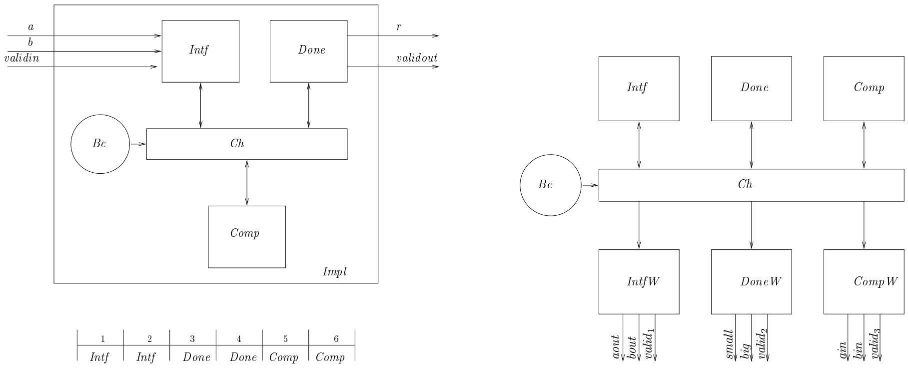
Fig. 1. GCD Specification

We consider a design that computes the Greatest Common Divisor (GCD) of two numbers. We will start with a synchronous specification shown in Figure 1(a). Given two inputs a and b, the module *GCDSp1* computes the GCD of a and b and places the result in the output r. The boolean input *validin* asserts that the inputs are valid in the current round and the boolean output *validout* asserts that the output is valid in the current round. Module *GCDSp1* operates synchronously, with a delay of one round. If inputs a and b are given in the current round, then the output is available at r in the next round.

We refine our specification and add more spatial and temporal detail on how the GCD is computed. We use Euclid's algorithm to compute the GCD:

GCD (a,b)
{Given positive non-zero integers a and b, compute $GCD(a, b)$ }
(1) **if** $(a = b)$ **return** (a);
(2) **if** $((a = 1)$ or $(b = 1))$ **return** (1);
(3) $small := \mathbf{min}(a,\ b)$;
(4) $big := \mathbf{max}(a,\ b)$;
(5) **return** $(GCD(small, big - small))$;

The resulting refinement *GCDSp2* (Figure 1(b)) has three modules: *IntfS*, *DoneS*, and *CompS*. Given two numbers, the *DoneS* module decides if the GCD is computed trivially (if the numbers are equal, or one of the numbers is 1). If so,



(a) Implementation

(b) Implementation with witness modules

Fig. 2. GCD Implementation

it sends the result, otherwise, it resends the numbers in increasing order. Suppose *small* and *big* are sent by the *DoneS* module. The *CompS* module responds by sending *small* and $big - small$. The *IntfS* module takes data inputs from both *CompS* and the environment and feeds the data to the *DoneS* module. The modules *IntfS*, *DoneS*, and *CompS* communicate with each other using point-to-point communication links. Valid bits ($valid_1$, $valid_2$ and $valid_3$) are used to validate the presence of meaningful data on these links. For instance, if *IntfS* wants to send two numbers to *DoneS*, it places the numbers in *aout* and *bout*, and sets $valid_1$ to TRUE. Each of these communications is assumed to complete in one round.

While *GCDSp1* requires only one round to compute the *GCD*, the module *GCDSp2* requires multiple rounds depending on the data inputs. We add an additional variable *inprogress* in module *GCDSp2* and set it to TRUE whenever a *GCD* computation is in progress. Using this variable and the *Sample* operator in Section 4, we will formally state how *GCDSp2* refines *GCDSp1*.

Our final level of refinement uses a single physical broadcast channel for communication between the modules. Time-division multiplexed access (TDMA) is used to share the channel. Communication in the channel is conducted in units called *frames*. A frame is divided, in time, into several *time-slots*. Each module is allocated one or more time-slots to send data. There is a beacon module *Bc*, that signals the beginning of a frame. Each module has its local counter that is synchronized on the *Bc* module's signal. Once the frame starts, each module sends data in its allocated time-slots. A valid bit sent on the channel indicates if the data being sent in the current time-slot is valid. The allocation of time-slots to modules is done statically at configuration time, and stays fixed thereafter. Thus, every module knows the identity of the sender in each time-slot. Figure 2(a) shows the block digram of the implementation *Impl*. In our example, a frame is divided into 6 slots. The figure also shows the allocation of time-slots within the frame to individual modules —the first two time-slots are given to the *Intf* module, the next two to the *Done* module, and the last two to the *Comp* module. The *Intf*, *Done*, and *Comp* modules are intended to have the same functionality as the specification modules *IntfS*, *DoneS*, and *CompS* from *GCDSp2*. However, while the communication between modules in *GCDSp2* happens in a single round through point-to-point links, the communication between modules in *Impl* is through a shared channel, and takes several rounds. Let *sync* be a variable of the module *Bc* that is set to TRUE whenever the *Bc* module sends the synchronizing signal. We will use *sync* and the *Sample* operator to relate *Impl* to *GCDSp2* in Section 4.

3 Transition Constraints and Modules

Transition constraints. Reactive Modules is a formalism for the modular description of systems with heterogeneous components. The definition of reactive modules can be found in [AH96]. In this paper, we generalize modules to transition constraints. A *transition constraint* is a temporal constraint on a set of variables. The state of a transition constraint is determined by values of two kinds of variables, namely, *observable variables* and *private variables.* The state of a transition constraint changes in a sequence of rounds. The first round is called the *initial round*, and determines initial values for all variables. Each subsequent round is called an *update round*, and determines new values for all variables. A sequence of states such that the first state results from an initial round, and each successive state in the sequence results from the previous state from an update round is called a *trajectory.* The projection of a trajectory to the observable variables is called a *trace.* The semantics of a transition constraint is its set of traces.

A state of a transition constraint A is a valuation for all its variables. We use primed variables to denote the new value in a round, and unprimed variables to denote the old value. Formally, the transition constraint A contains two predicates, namely, an *initial predicate* and *update predicate.* The initial predicate is a boolean function over the primed variables of A. A state s of A is *initial* if it satisfies the initial predicate of A. The update predicate is a boolean function over both the primed and unprimed variables of A. Given two states s and t, we write $s \rightarrow_A t$ if the update predicate of A evaluates to TRUE, when its unprimed variables are assigned values from s and its primed variables are assigned values from t. A *trajectory* of A is a finite sequence $s_0, \ldots, s_n$ of states such that (1) s_0 is an initial state of A, and (2) for $i \in \{0, 1, \ldots, n-1\}$, we have $s_i \rightarrow_A s_{i+1}$. The states that lie on trajectories are called *reachable.* An *observation* of A is a valuation for the observable variables of A. If s is a valuation to a set of variables, we use $[s]_A$ to denote the valuation restricted to the observable variables of A. For a state sequence $\overline{s} = s_0, \ldots, s_n$, we write $[\overline{s}]_A = [s_0]_A, \ldots, [s_n]_A$ for the corresponding observation sequence. If $\overline{s}$ is a trajectory of A, then the projection $[\overline{s}]_A$ is called a *trace* of A. Sometimes there is a need to effect a transition constraint only for a fixed number of update steps. We use A^τ to denote the transition constraint with the same variables as A, which restricts their valuations only up to τ update rounds. Formally, a state sequence $\overline{s} = s_0, \ldots, s_n$ is a trajectory of A^τ if (1) $n \leq \tau$ and $\overline{s}$ is a trajectory of A, or (2) $n > \tau$ and $s_0, \ldots, s_\tau$ is a trajectory of A. Note that if $\tau < 0$, then every state sequence is a trajectory of A^τ.

Modules. Modules are a special class of transition constraints with two special properties: (1) differentiation between inputs and outputs, and (2) executable non-blocking semantics. The observable variables of a module are partitioned into *external variables* and *interface variables.* External variables are updated by the environment and can be read by the module, and interface variables are updated by the module and can be read by the environment. The interface and private variables of a module are called *controlled variables.*

The state of a reactive module changes again in a sequence of rounds. For the external variables, the values in the initial and update rounds are left unspecified (i.e., chosen nondeterministically). For the controlled variables, the values in the initial and update rounds are specified by (possibly nondeterministic) guarded commands. A formal description of modules can be found in [AH96].

Parallel composition. The composition operation combines two transition constraints into a single transition constraint whose behavior captures the interaction between the two components. Two transition constraints A and B are *compatible* if their sets of private variables is disjoint. Given two compatible transition constraints A and B, the *composition* $A\|B$ is a transition constraint. The observable variables of $A\|B$ are the union of the observable variables of A and B. The private variables of $A\|B$ are the (disjoint) union of the private variables of A and B. The initial predicate of $A\|B$ is the conjunction of the initial predicates of A and B, and the update predicate of $A\|B$ is the conjunction of the update predicates of A and B.

Refinement. The notion that two transition constraints describe the same system at different levels of detail is captured by the refinement relation between transition constraints. The transition constraint B is *refinable* by transition constraint A if every observable variable of B is an observable variable of A. The transition constraint A *refines* the transition constraint B, written $A \preceq B$, if (1) B is refinable by A, and (2) for every trajectory $\overline{s}$ of A, the projection $[\overline{s}]_B$ is a trace of B.

4 The Sample Operator

Let A be a transition constraint and φ be a predicate on the primed and unprimed observable variables of A. We define $B = (\textit{Sample } A \textit{ at } \varphi)$ to be a transition constraint. The private and observable variables of B are the private and observable variables of A. The initial predicate of B is equal to the initial predicate of A. The update predicate of B is TRUE at the pair of states (s, t) iff there is a sequence of states $s_0, s_1, \ldots, s_n$, such that (1) $s = s_0$, (2) $t = s_n$, (3) $s_i \rightarrow_A s_{i+1}$ for $i = 0, 1, \ldots, n-1$, and (4) $\varphi(s_i, s_{i+1})$ is FALSE for $i = 0, 1, \ldots, n-2$ and TRUE for $i = n-1$. Informally, B updates from state s to t if there exists a sequence of rounds of A starting at s and ending at t, such that the final round satisfies φ, and none of the intermediate rounds satisfy

φ. Given a trajectory of A, we use the term *sampling instants* to refer to the instants in the trajectory where φ is true.

We note that the *Sample* operator differs from the *Next* operator of [AH96] in two ways: (1) *Sample* operates on transition constraints, whereas *Next* operates on modules, and (2) *Sample* does not constrain the environment between sampling instants, whereas *Next* constrains the environment to not change between sampling instants. In the rest of this paper, whenever we write $B = (\mathit{Sample}\ A\ \mathit{at}\ \varphi)$, we assume that φ is a predicate on the primed and unprimed observable variables of A.

Example. Recall the *GCD* computation example from Section 2. The module *GCDSp1* is the high-level specification. The intermediate-level specification and implementation modules are composed as follows:

$$\begin{aligned} \mathit{GCDSp2} &= \mathit{IntfS} \| \mathit{DoneS} \| \mathit{CompS} \\ \mathit{Impl} &= \mathit{Bc} \| \mathit{Ch} \| \mathit{Intf} \| \mathit{Done} \| \mathit{Comp} \end{aligned}$$

We wish to relate the intermediate specification *GCDSp2* to the high level specification *GCDSp1*. Recall that *GCDSp1* computes *GCD* in one round, whereas *GCDSp2* takes multiple rounds. Also recall that *inprogress* is a variable of *GCDSp2* that is set to TRUE when *GCDSp2* is doing the *GCD* computation. If we sample the behaviors of *GCDSp2* during the instances where *inprogress* is FALSE, the sampling should conform to the behaviors allowed by *GCDSp1*. Using the *Sample* operator, we can express this requirement as:

$$(\mathit{Sample}\ \mathit{GCDSp2}\ \mathit{at}\ (\neg \mathit{inprogress}')) \preceq \mathit{GCDSp1}$$

We also wish to relate *GCDSp2* to the final implementation *Impl*. Recall that every communication in *GCDSp2* happens in a single round, whereas every communication in *Impl* takes several rounds (as many rounds it takes to transmit a frame) to complete. Recall that the *Bc* module has a variable *sync* that is set to TRUE when it sends the synchronization signal. Though *Impl* and *GCDSp2* operate at different time scales, if we consider any trace of *Impl* and sample only the instants where *sync* is TRUE, the resulting subsequence should be a trace of *GCDSp2*. Using the *Sample* operator, we can express this requirement as:

$$(\mathit{Sample}\ \mathit{Impl}\ \mathit{at}\ \mathit{sync}') \preceq \mathit{GCDSp2}$$

Properties of Sample. The following proposition asserts the distributivity of *Sample* with respect to the parallel-composition operator.

Proposition 1. *[Distributivity of Sample] If A, B are any two transition constraints and φ is a predicate on the observable variables common to A and B, then* $(\mathit{Sample}\ (A\|B)\ \mathit{at}\ \varphi) \preceq (\mathit{Sample}\ A\ \mathit{at}\ \varphi) \parallel (\mathit{Sample}\ B\ \mathit{at}\ \varphi)$.

In practice, the traces of $(\mathit{Sample}\ A\ \mathit{at}\ \varphi) \parallel (\mathit{Sample}\ B\ \mathit{at}\ \varphi)$ form a large superset of the traces of $(\mathit{Sample}\ (A\|B)\ \mathit{at}\ \varphi)$. It is desirable to constrain the observable variables of A and B while distributing the *Sample* operator over parallel composition. We can strengthen the above proposition in the presence of suitable transition constraints T_A and T_B on the observable variables of A and B, respectively. The resulting proposition, given below, will be used in the next section to carry out assume-guarantee reasoning between different time scales.

Proposition 2. *[Constrained distributivity of Sample] Consider transition constraints A, B, T_A and T_B, such that every variable of T_A is an observable variable of A, and every variable of T_B is an observable variable of B. If $A \| B \preceq T_A \| T_B$ and φ is a predicate on the observable variables common to A and B, then (Sample $(A \| B)$ at φ) $\preceq$ (Sample $(A \| T_A)$ at φ) $\|$ (Sample $(B \| T_B)$ at φ).*

5 Refinement Checking Methodology

We generalize the methodology for assume-guarantee style refinement checking given in [HQR98] to accommodate the *Sample* operator.

Witness modules. The problem of checking if $A \preceq B$ is PSPACE-hard in the state space of B. However, the refinement check is simpler in the special case in which all variables of B are observable. The transition constraint A is *projection refinable* by transition constraint B if (1) B is refinable by A, and (2) B has no private variables. If B is projection refinable by A, then every variable of B is observable in both B and A. Therefore, checking if $A \preceq B$ reduces to checking if for every trajectory $\overline{s}$ of A, the projection $[\overline{s}]_B$ is a *trajectory* of B. This can be done by a transition-invariant check [HQR98], whose complexity is linear in the state spaces of both A and B. If B is projection refinable by A, then $A \preceq B$ iff (1) if s is an initial state of A, then $[s]_B$ is an initial state of B, and (2) if s is a reachable state of A and $s \rightarrow_A t$, then $[s]_B \rightarrow_B [t]_B$.

Suppose that B is refinable by A, but not projection refinable. This means that there are some private variables in B. Define B^u to be the transition constraint obtained by making every private variable of B an observable variable. If we compose A with a module W whose observable variables include the private variables of B, then B^u is projection refinable by the composition $A \| W$. Moreover, if W does not constrain any observable variables of A, then $A \| W \preceq B^u$ implies $A \preceq B$ (in fact, A is simulated by B). Such a module W is called a *witness* to the refinement $A \preceq B$. We require W to be a module, because we need to enforce that W does not constrain the observable variables of A, and that it does not deadlock. In order to check refinement, it is sufficient to first find a witness module and then check projection refinement. If the sample operator needs to be applied to the implementation to relate it to the specification, the witness could be composed either before or after applying the *Sample* operator.

Proposition 3. *[Sampled Witnesses] Consider two transition constraints A and B, and a predicate φ on the variables of A such that B is refinable by A. Let W be a module such that the interface variables of W include the private variables of B, and are disjoint from the observable variables of A. Then (1) B^u is projection refinable by (Sample $(A \| W)$ at φ), and (2) (Sample $(A \| W)$ at φ) $\preceq$ B^u implies (Sample A at φ) $\preceq B$.*

Proposition 3 is a generalization of a proposition from [HQR98]. The latter can be obtained by setting φ to TRUE in Proposition 3.

Assume-guarantee reasoning. The state space of a transition constraint may be exponential in the size of its description. Consequently, even checking projection refinement may not be feasible. However, typically both the implementation

A and the specification B consist of the parallel composition of several transition constraints, in which case it may be possible to reduce the problem of checking if $A \preceq B$ to several subproblems that involve smaller state spaces. The assume-guarantee rules allows us to conclude $A \preceq B$ as long as each component of the specification B is refined by the corresponding components of the implementation B within a suitable environment. The apparent circularity in the environment assumptions is resolved by an induction over time. If we operate with modules, the acyclic await dependencies of legal modules breaks this circularity. Since our purpose in this paper is to carry out assume-guarantee reasoning with the *Sample* operator, which gives transition constraints, not modules, we impose a well-founded order on the specification components [McM98] to break the circularity.

Proposition 4. *[Assume-Guarantee [McM98]] Let* $A = A_1 \| A_2 \| \ldots \| A_n$ *and* $B = B_1 \| B_2 \| \ldots \| B_m$ *be transition constraints. Let* $\prec$ *be a well-founded order on the components of* B*, let* $Z(B_i) = \{B_j | B_j \prec B_i\}$*, and let* $Z^C(B_i) = \{B_j | B_j \notin Z(B_i)\}$*. For each* B_i*, let* C_i *be some composition of transition constraints from* A*, let* D_i *be some composition of transition constraints from* $Z(B_i)$*, and let* E_i *be some composition of transition constraints from* $Z^C(B_i)$*. If* $C_i^\tau \| D_i^\tau \| E_i^{\tau-1} \preceq B_i^\tau$ *for all* $1 \leq i \leq m$ *and* $\tau \in \mathbb{N}$*, then* $A \preceq B$*.*

Proposition 4 produces proof obligations of the form $C_i^\tau \| D_i^\tau \| E_i^{\tau-1} \preceq B_i^\tau$. For each specification component B_i, the corresponding implementation component that is intended to implement the functionality specified by B_i is C_i , and the constraining environments are D_i and E_i. This obligation can be discharged by a reachability computation on $C_i \| D_i \| E_i$. At each stage of the reachability one has to merely check that every transition possible in $C_i \| D_i$ is also allowable in B_i. Note that while the transition constraint E_i is used to constrain the reachable states of $C_i \| D_i$, it is not used to constrain the transition invariant check at each step.

While discharging the obligation for B_i according to Proposition 4, we would like to keep the state spaces to be small. Thus, it is preferable to choose most components of the constraining environment from the specification. Unfortunately, due to lack of detail, the specification does not have sufficiently abstract definitions of internal variables of the implementation. For all transition constraints A, B, and F, if $A \preceq B \| F$ and B is refinable by A, then $A \preceq B$. Thus, we can arbitrarily "enrich" the specification by composing it with new transition constraints. Before applying the assume-guarantee rule, we may add components to the specification and prove $A \preceq B \| F_1 \| \cdots \| F_k$ instead of $A \preceq B$. The new transition constraints $F_1, \ldots, F_k$ are called *abstract constraints*, and they usually give high-level descriptions for some implementation variables, in order to provide a sufficient supply of abstract components while applying Proposition 4. While witness modules are introduced "on the left" of a refinement relation, abstract constraints are introduced "on the right."

Suppose the left side of a refinement relation is of the form $Sample(A \| B)$ *at* φ. It is not directly possible to apply the assume-guarantee rule from Proposition 4 in such cases. However, we can distribute the *Sample* operator with respect to parallel composition using Proposition 1. In practice, Proposition 1 tends to provide abstractions that are too coarse to be useful. To see why, imagine A and B as modules, each constraining the other's inputs. By distributing the *Sample* inside the parallel composition, B is allowed to constrain the inputs to A only at the sampling instants. The inputs to A are essentially unconstrained between sampling instants. Symmetrically, the inputs to B are constrained at sampling instants by A and left unconstrained between sampling instants. In several common situations, the interactions between A and B can be orthogonalized into (1) functionality, and (2) timing of the communication protocol used for the interaction. The functionality determines values at the sampling instants, whereas the timing determines how these values propagate between sampling instants.

Suppose T_A and T_B are transition constraints that specify how the inputs to A and B behave between sampling instants. Then, we can use Proposition 2 to distribute the *Sample* operator, while constraining the inputs to A by T_A and the inputs to B by T_B. Thus, we get the following generalization of the assume-guarantee rule, with the *Sample* operator.

Theorem 1. *[Assume-Guarantee with Sample] Let* $A = A_1 \| A_2 \| \ldots \| A_n$ *and* $B = B_1 \| B_2 \| \ldots \| B_m$ *be transition constraints. Let* T_i *be a transition constraint on the observable variables of* A_i*, for* $1 \leq i \leq n$*. Let* $\prec$ *be a well-founded order on the components of* B*, and let* T*,* Z *and* Z^C *be defined as follows:* $T = \{T_1, \ldots, T_n\}$*,* $Z(B_i) = \{B_j | B_j \prec B_i\}$*, and* $Z^C(B_i) = \{B_j | B_j \notin Z(B_i)\}$*. For each* B_i*, let* C_i *be some composition of transition constraints from* A*, let* U_i *be some composition of transition constraints from* T*, let* D_i *be some composition of transition constraints from* $Z(B_i)$*, and let* E_i *be some composition of transition constraints from* $Z^C(B_i)$*. If* $A \preceq T_1 \| \ldots \| T_n$*, and* ψ *is any predicate such that* $(\mathit{Sample}\ (C_i \| U_i)\ \mathit{at}\ \psi)^\tau \| D_i^\tau \| E_i^{\tau-1} \preceq B_i^\tau$ *for all* $1 \leq i \leq m$ *and* $\tau \in \mathbb{N}$*, then* $(\mathit{Sample}\ A\ \mathit{at}\ \psi) \preceq B$*.*

In the above theorem, note that the antecedent $A \preceq T_1 \| \ldots \| T_n$ can itself be discharged by traditional assume-guarantee reasoning (Proposition 4).

6 Refinement Proof

Recall the high-level specification *GCDSp1*, intermediate specification *GCDSp2*, and implementation *Impl* from the previous sections. As stated in Section 4, the refinements we would like to verify are:

$$(\mathit{Sample}\ \mathit{GCDSp2}\ \mathit{at}\ (\neg \mathit{inprogress}')) \preceq \mathit{GCDSp1}$$
$$(\mathit{Sample}\ \mathit{Impl}\ \mathit{at}\ \mathit{sync}') \preceq \mathit{GCDSp2}$$

In this section, we will focus on how to carry out the second refinement, which relates the intermediate specification *GCDSp2* to the implementation *Impl*. We first observe that *GCDSp2* is not projection refinable by *Impl*, due to the presence of private variables in *GCDSp2* that represent point-to-point communication channels. The module *Impl* has a single channel that is shared by all modules using TDMA. The specification, while more abstract in time, provides individual point-to-point channels for communication. Each round of *GCDSp2* corresponds to multiple rounds of *Impl*, during which a frame is transmitted. It is possible to relate the values that appear on the shared implementation channel, at particular time-slots during the communication of a frame, to values that appear on particular point-to-point channels in the specification. For instance, the values of the specification variables *aout* and *bout* at the end of each round are expected to be equal to the values occurring in time-slots 1 and 2 of the frame. We can write a witness module *IntfW* that looks at the implementation channel during the transmission of the frame, collects the values at time-slots 1 and 2, and assigns them to *aout* and *bout*, respectively. If the values in time-slots 1 and 2 are valid, then $valid_1$ is set to TRUE. Further, the values assigned to *aout*, *bout*, and $valid_1$ are retained till the end of the frame. Similar witness modules *DoneW* and *CompW* can be written. All these witnesses take inputs from the channel as shown in Figure 2(b). Let *ImplW* be the module given by

$$ImplW = Impl \| IntfW \| DoneW \| CompW.$$

Note that the witnesses do not interfere with the channel —they merely observe the values on the channel. Due to Proposition 3, it suffices to check that $(Sample\ ImplW\ at\ sync') \preceq GCDSp2$. Recall $GCDSp2 = IntfS \| DoneS \| CompS$. We use the order $DoneS \prec CompS \prec IntfS$ and apply Theorem 1. Let us consider the component *CompS* of *GCDSp2*. The component of *Impl* that is intended to implement the functionality of *CompS* is *Comp*. We wish to check if:

$$(Sample\ Comp\ at\ sync')^{\tau} \| DoneS^{\tau} \preceq CompS^{\tau}.$$

This check fails because the outputs of module *CompS*, namely, *ain*, *bin* and $valid_3$ are not present in module *Comp*. Adding the witness module and appropriately constraining it, we obtain the obligation:

$$(Sample\ (Comp \| CompW \| Ch \| Bc)\ at\ sync')^{\tau} \preceq CompS^{\tau}.$$

This still fails, because we have not constrained the inputs of *Comp*. In this obligation, the specification *CompS* looks at the inputs *small*, *big*, and $valid_2$ in every round and produces corresponding outputs *ain*, *bin* and $valid_3$ in the next round. The implementation *Comp* anticipates two values at time-slots 3 and 4 (which correspond to *small* and *big*, respectively) of every frame. If these inputs are valid, then *Comp* generates values in time-slots 5 and 6 (which correspond to *ain* and *bin* respectively) of the next frame. The module *Comp* makes the following assumptions: (1)the inputs are available at time-slots 3 and 4, (2)either both inputs are available at a given frame, or none of the inputs are available, and (3)if both inputs are available, the first input (from time-slot 3) is smaller than the second input (from time-slot 4). In our refinement obligation, the inputs to *Comp* have to be constrained both at the sampling instants and between sampling instants, in order to satisfy the above assumptions. The specification component *DoneS* supplies the constraint at sampling instants, which ensures that assumptions 2 and 3 are satisfied. The timing assumption *DoneW* constrains the inputs between sampling instants and ensures that assumption 1 is satisfied. Thus we get the proof obligation

$$(Sample\ (Comp \| CompW \| Ch \| Bc \| DoneW)\ at\ sync')^{\tau} \| DoneS^{\tau} \preceq CompS^{\tau}.$$

Similarly, we can verify the correctness of modules *Done* and *Intf* separately. The complete refinement proof, which is a direct application of Theorem 1, uses the ordering $DoneS \prec CompS \prec IntfS$:

$$\frac{\begin{array}{ll} (\mathit{Sample}\ (\mathit{Done}\|\mathit{DoneW}\|\mathit{Ch}\|\mathit{Bc}\|\mathit{IntfW})\ \mathit{at}\ \mathit{sync}')^{\tau} & \preceq \mathit{DoneS}^{\tau} \\ (\mathit{Sample}\ (\mathit{Comp}\|\mathit{CompW}\|\mathit{Ch}\|\mathit{Bc}\|\mathit{DoneW})\ \mathit{at}\ \mathit{sync}')^{\tau}\|\mathit{DoneS}^{\tau} & \preceq \mathit{CompS}^{\tau} \\ (\mathit{Sample}\ (\mathit{Intf}\|\mathit{IntfW}\|\mathit{Ch}\|\mathit{Bc}\|\mathit{CompW})\ \mathit{at}\ \mathit{sync}')^{\tau}\|\mathit{CompS}^{\tau}\|\mathit{DoneS}^{\tau} & \preceq \mathit{IntfS}^{\tau} \end{array}}{\begin{array}{ll} (\mathit{Sample}\ (\mathit{ImplW})\ \mathit{at}\ \mathit{sync}') & \preceq \mathit{GCDSp2} \end{array}}$$

Each of the obligations above the line involves a single implementation component, possibly along with specification components, witnesses and abstract constraints to constrain the inputs. They can be automatically discharged by Mocha. We thus conclude that $(\mathit{Sample}\ \mathit{Impl}\ \mathit{at}\ \mathit{sync}') \preceq \mathit{GCDSp2}$.

7 VGI Processor Verification

The approach described in this paper has been used in a case study [HLQR99] to verify a large parallel DSP processor. The VGI chip [STUR98] is an array of DSP processors designed to be part of a system for web-based image processing [SR97]. The VGI chip contains a total of 96 processors and has approximately 6M transistors. Of the 96 processors, 64 are 3-stage pipelined compute processors. Each compute processor has about 30,000 logic gates. Data is communicated between the processors by means of FIFO queues. No assumptions are made about the relative speeds at which data is produced and consumed in the processors. Hence, to transfer data reliably, an elaborate handshake mechanism is used between the sender and the receiver. In addition, the interaction between the control of the pipeline and the control of the communication unit is quite complex. The details of the VGI processor verification can be found in [HLQR99]. Here, we only discuss the role the *Sample* operator in the verification of VGI.

The implementation has a clock signal `clk` with activity on both the `HIGH` and `LOW` phases in different parts of the design. For instance, in the execute phase of the pipeline a bus carries an operand when `clk` is `HIGH` and the result when `clk` is `LOW`. But the specification does not mention `clk` at all. In fact, the whole computation happens in just one step. Thus, one round of the specification is equal to two rounds of the implementation, one with `clk` = `HIGH` and one with `clk` = `LOW`. Therefore, we sample the implementation whenever `clk` is low and check if the sampled behavior is present in the specification.

In the remainder of this section, we use φ to refer to `clk` = `LOW`. Our goal is to verify that an arbitrary network of compute processors implements its corresponding ISAs(instruction set architectures), using refinement checking. Let $P_1, P_2, \ldots, P_n$ be the compute processors, and let $Q_1, Q_2, \ldots, Q_n$ be their respective ISAs. The verification problem is to check that

$$\mathit{Sample}\ (P_1\|P_2\|\cdots\|P_n)\ \mathit{at}\ \varphi \preceq (Q_1\|Q_2\|\cdots\|Q_n).$$

For the correct functioning of a processor it is essential that all input signals change only when `clk` is `HIGH`. Let T_i be a module that says that all input signals of P_i change only when `clk` is `HIGH`. We use Theorem 1 to decompose this proof as follows:

$$\frac{\begin{array}{c}(\mathit{Sample}(P_i\|T_i)\ \mathit{at}\ \varphi) \preceq Q_i \text{ for all } 1 \leq i \leq n \\ P_1\|P_2\|\cdots\|P_n \preceq T_1\|T_2\|\cdots\|T_n\end{array}}{\mathit{Sample}(P_1\|P_2\|\cdots\|P_n)\ \mathit{at}\ \varphi \preceq Q_1\|Q_2\|\cdots\|Q_n}$$

The second antecedent says that the inputs of any processor in the network change only when `clk` is `HIGH`. Since any input to a processor has to be the output of some other processor, this antecedent can be discharged easily by proving that for all $1 \leq i \leq n$, the outputs of P_i change only when `clk` is `HIGH`. This is an easy proof local to each processor. In the first antecedent, there are n symmetric proof obligations, one for each P_i. Each P_i can be in any one of a finite number of configurations. Moreover, if P_j and P_k are in the same configuration, then the jth and kth proofs are identical except for variable renaming. In this way, we decompose the proof of a 64 processor network to proofs about individual processor configurations that have 800 latches each. This is still beyond the scope of monolithic model checking. We use Theorem 1 again, to decompose the proof for a single processor configuration into about 35 proof obligations and check them using Mocha. None of the individual lemmas require more than a few minutes on a 625 MHz DEC Alpha 21164. We found and fixed several subtle bugs that were unknown to the designers. Most bugs were in the interaction between the pipeline and the communication protocol.

References

[AH96] R. Alur and T.A. Henzinger. Reactive modules. In *Proc. 11th Annual Symposium on Logic in Computer Science*, pp 207–218, 1996.

[AHM+98] R. Alur, T.A. Henzinger, F.Y.C. Mang, S. Qadeer, S.K. Rajamani, and S. Tasiran. Mocha : Modularity in model checking. In *CAV 98: Computer Aided Verification*, Springer LNCS 1427, pp 521–525, 1998.

[AHR98] R. Alur, T.A. Henzinger, and S.K. Rajamani. Symbolic exploration of transition hierarchies. In *TACAS 98: Tools and Algorithms for Construction and Analysis of Systems*, Springer LNCS 1384, pp 330–344, 1998.

[AL95] M. Abadi and L. Lamport. Conjoining specifications. *ACM Transactions on Programming Languages and Systems*, 17(3):507–534, 1995.

[HLQR99] T. A. Henzinger, X. Liu, S. Qadeer, and S. K. Rajamani. Formal specification and verification of a dataflow processor array. Technical Report M99/14, ERL, University of California, Berkeley, 1999.

[HQR98] T.A. Henzinger, S. Qadeer, and S.K. Rajamani. You assume, we guarantee: methodology and case studies. In *CAV 98: Computer Aided Verification*, Springer LNCS 1427, pp 440–451, 1998.

[Kur94] R.P. Kurshan. *Computer-aided Verification of Coordinating Processes.* Princeton University Press, 1994.

[McM97] K.L. McMillan. A compositional rule for hardware design refinement. In *CAV 97: Computer-Aided Verification*, Springer LNCS 1254, pp 24–35, 1997.

[McM98] K.L. McMillan. Verification of an implementation of Tomasulo's algorithm by compositional model checking. In *CAV 98: Computer-Aided Verification*, Springer LNCS 1427, pp 110-121, 1998.

[SR97] V.P. Srini and J.M. Rabaey. An architecture for web-based image processing. In *Proc. SPIE Conference 3166*, pp 90–101, 1997.

[Sta85] E.W. Stark. A proof technique for rely/guarantee properties. In *Proc. 5th Conference on Foundations of Software Technology and Theoretical Computer Science*, Springer LNCS 206, pp 369–391, 1985.

[STUR98] V.P. Srini, J. Thendean, S.Z. Ueng, and J.M. Rabaey. A parallel DSP with memory and I/O processors. In *Proc. SPIE Conference 3452*, pp 2–13, 1998.