Automated Refinement Checking for Asynchronous Processes

Rajeev Alur, Radu Grosu, and Bow-Yaw Wang

Department of Computer and Information Science
University of Pennsylvania
200 South 33rd Street
Philadelphia, PA 19104
Email: alur,grosu,bywang@cis.upenn.edu
URL: www.cis.upenn.edu/~alur,grosu,bywang

Abstract. We consider the problem of refinement checking for asynchronous processes where refinement corresponds to stutter-closed language inclusion. Since an efficient algorithmic solution to the refinement check demands the construction of a *witness* that defines the private specification variables in terms of the implementation variables, we first propose a construction to extract a synchronous witness from the specification. This automatically reduces individual refinement checks to reachability analysis. Second, to alleviate the state-explosion problem during search, we propose a reduction scheme that exploits the visibility information about transitions in a recursive manner based on the architectural hierarchy. Third, we establish compositional and assume-guarantee proof rules for decomposing the refinement check into subproblems. All these techniques work in synergy to reduce the computational requirements of refinement checking. We have integrated the proposed methodology based on an enumerative search in the model checker Mocha. We illustrate our approach on sample benchmarks.

1 Introduction

Refinement checking is the problem of analyzing a detailed design, called the *implementation*, with respect to an abstract design, called the *specification*. Refinement checking is useful for detecting logical errors in designs in a systematic manner, and offers the potential of developing systems formally by adding details in a stepwise manner. Our notion of refinement is based on language inclusion, and due to its high computational complexity, traditionally refinement proofs are done manually or using theorem provers[1]. A recent promising approach to *automated* refinement checking combines assume-guarantee reasoning with BDD-based symbolic search routines [21, 12, 13], and has been successfully applied to synchronous hardware designs such as pipelined processors [22] and a

[1] Note that stronger notions such as bisimulation and simulation are used in many process algebras, and are supported by automated tools [23, 7, 14, 26].

VGI chip [11]. In this paper, we develop the foundations and heuristics for automated refinement checking of *asynchronous* processes, and report on several case studies.

The standard notion of refinement is inclusion of trace languages: every observable behavior of the implementation must also be a possible behavior of the specification. In the asynchronous setting such as shared memory multiprocessors, asynchronous circuits, and distributed processes communicating by messages, there is no notion of global time, and the speeds of the processes are independent. Consequently, behaviors that differ from each other only modulo stuttering (repetition of the same observation) need to be considered identical, and the appropriate notion of refinement is inclusion of *stutter-closed* languages: $P < Q$ iff every trace of P is stutter-equivalent to some trace of Q [25, 2, 18].

Given two processes P and Q, checking whether $P < Q$ holds, is computationally hard: if Q is nondeterministic then it must be determinized, which requires subset construction, and furthermore, to account for stuttering, an ε-closure construction must be applied, which requires computation of the transitive closure of the transition relation. Both these problems disappear if the specification process Q has no private state. Then, as in the synchronous case, it suffices to establish that every reachable transition of P has a corresponding transition in Q, and this can be done by an on-the-fly search routine that can report counter-examples. On the other hand, when the specification has private variables, the classical approach is to require the user to provide a definition of the private variables of the specification in terms of the implementation variables (this basic idea is needed even for manual proofs, and comes in various formalizations such as refinement maps [1], homomorphisms [17], forward-simulation maps [19], and witness modules [12, 21]). Thus, the refinement check $P < Q$ reduces to $P \| W < Q$, where W is the user-supplied witness for private variables of Q. In our setting of asynchronous processes, it turns out that the witness W itself should not be asynchronous (that is, for asynchronous W, $P \| W < Q$ typically does not hold). This implies that the standard trick of choosing Q itself as a witness, used in many of the case studies reported in [22, 11], does not work in the asynchronous setting. As a heuristic for choosing W automatically, we propose a construction that transforms Q to $Eager(Q)$, which is like Q, but takes a stuttering step only when all other choices are disabled. This construction is syntactically simple, and as our case studies demonstrate, turns out to be an effective way of automating witness construction. The complexity of the resulting check, is then, proportional to the product of P and Q.

The second component of the proposed method is a heuristic for on-the-fly search based on compressing unobservable transitions in a hierarchical manner. This is an extension of our earlier work on efficient invariant verification [6]. The basic idea is to describe the implementation P in a hierarchical manner so that P is a tree whose leaves are atomic processes, and internal nodes compose their children and hide as many variables as possible. The basic reduction strategy, proposed by many researchers, is simple: while computing the successors of a state of a process, apply the transition relation repeatedly until a shared variable is accessed. This is applicable since changes to private state are treated as stuttering steps. The novelty is in applying the reduction in a recursive manner exploiting the hierarchical structure. Our strategy is easy to implement, and gives significant reductions in space and time requirements, particularly for well-structured systems such as rings and trees.

The last component of our methodology is an *assume guarantee* principle for the stutter-closed refinement in the context of our modeling language of *reactive modules* [4]. Our assume guarantee principle asserts that to prove $P_1\|P_2 < Q_1\|Q_2$, it suffices to establish separately $P_1\|Q_2 < Q_1$ and $Q_1\|P_2 < Q_2$. This principle, similar in spirit to many previous proposals [27, 2, 4, 10, 21], reduces the verification of a composition of implementation components to individual components, but verifies an individual component only in the context of the specifications of the other components. Our first two techniques are used to check individual components afterwards.

We have incorporated our methodology using an enumerative search engine in the new implementation of the model checker MOCHA (see [5] for a description of the first release). This version is implemented in Java, and supports an extensive GUI with a proof assistant that allows the user to select refinement goals, and generates subgoals via compositional and assume-guarantee rules. The counter-examples generated by the refinement checker are displayed by a simulator in the message-sequence-chart like format. The tool is available at `www.cis.upenn.edu/~mocha`

The case studies reported in this paper include (1) an assume-guarantee style proof relating two descriptions of alternating-bit protocol, (2) a refinement check of a tree-structured implementation of an n-way arbiter using 2-way elements (this illustrates the use of eager witnesses, and efficiency of the heuristic for hierarchical reduction), (3) a leader election protocol (illustrating the efficiency of the hierarchical reduction), and (4) ring of distributed mutual exclusion cells [20] (illustrating use of automatic witness construction, assume-guarantee proofs, and hierarchical reduction). While these examples have been analyzed previously by model checkers, prior studies have focussed on verifying temporal logic requirements, and in contrast, our tool checks (stutter-closed) refinement with respect to an abstract specification process.

2 Process Model

We start with the definition of processes. The model is a special class of *reactive modules* [4] that corresponds to asynchronous processes communicating via read-shared variables. A process is defined by the set of its variables, rules for initializing the variables, and rules for updating the variables. The variables of a process P are partitioned into three classes: *private* variables that cannot be read or written by other processes, *interface* variables that are written only by P, but can be read by other processes, and *external* variables that can only be read by P, and are written by other processes. Thus, interface and external variables are used for communication, and are called *observable* variables. The process controls its private and interface variables, and the environment controls the external variables. The separation between controlled and external variables is essential for the assume guarantee reasoning, and the separation between private and observable variables is essential for compressing internal transitions effectively. Once the variables are defined, the state space of the process is determined. A state is also partitioned into different components as the variables are, for instance, controlled state and external state. The initialization specifies initial controlled states, and the transition relation specifies how to change the controlled state as a function of the current state.

Definition 1. *A* process P *is a tuple* (X, I, T) *where*

- $X = (X_p, X_i, X_e)$ *is the (typed) variable declaration.* X_p*,* X_i*,* X_e *represent the sets of* private variables*,* interface variables *and* external variables *respectively. Define the* controlled variables $X_c = X_p \cup X_i$ *and the* observable variables $X_o = X_i \cup X_e$*;*
- *Given a set* X *of typed variables, a* state *over* X *is a function mapping variables to their values. Define* Q_c *to be the set of* controlled states *over* X_c *and* Q_e *the set of* external states *over* X_e*.* $Q = Q_c \times Q_e$ *is the set of states. We also define* Q_o *to be the set of* observable states *over* X_o*;*
- $I \subseteq Q_c$ *is the set of* initial states*;*
- $T \subseteq Q_c \times Q_e \times Q_c$ *is the* transition relation *with the property (called* asynchronous property*) that for any* $q \in Q_c$ *and any* $e \in Q_e$*,* $(q, e, q) \in T$*.*

■

Starting from a state q, a successor state is obtained by independently letting the process update its controlled state and the environment update the external state. The asynchronous property says that a process may idle at any step, and thus, the speeds of the process and its environment are independent.

Definition 2. *Let* $P = ((X_p, X_i, X_e), I, T)$ *be a process, and* q*,* q' *be states. Then* q' *is a* successor *of* q*, written* $q \longrightarrow^P q'$*, if* $(q[X_c], q[X_e], q'[X_c]) \in T$[2]*.*

■

Note that our model allows simultaneous updates by component processes, and thus, is different from the interleaving model (as used in SPIN [15], for instance). It is a special case of the model used by MOCHA , which supports a more general synchronous model in which the updates by the process and its environment can be mutually dependent in an acyclic manner. Modeling asynchrony within a synchronous model by a nondeterministic choice to stutter is a well-known concept [23].

In order to support structured descriptions, we would like to build complex processes from simpler ones. Two constructs, hide H in P and $P \| P'$ for building new processes are defined (we also support *instantiation*, or *renaming*, but it is not needed for the technical development in this paper). The hiding operator

[2] For a state q over variables X, and a subset $Y \subseteq X$, $q[Y]$ denotes the projection of q on the set Y.

makes interface variables inaccessible to other processes, and its judicious use allows more transitions to be considered internal.

Definition 3. *Let $P = ((X_p, X_i, X_e), I, T)$ be a process and $H \subseteq X_i$. Define the process* hide H in P *to be $((X_p \cup H, X_i \setminus H, X_e), I, T)$.* ■

The parallel composition operator allows to combine two processes into a single one. The composition is defined only when the controlled variables of the two processes are disjoint. This ensures that the communication is nonblocking, and is necessary for the validity of the assume guarantee reasoning.

Definition 4. *Let $P = ((X_p^P, X_i^P, X_e^P), I^P, T^P)$ and $Q = ((X_p^Q, X_i^Q, X_e^Q), I^Q, T^Q)$ be processes where $X_c^P \cap X_c^Q = \emptyset$. The* composition *of P and Q, denoted $P \| Q$, is defined as follows.*

- $X_p = X_p^P \cup X_p^Q$; $X_i = X_i^P \cup X_i^Q$; $X_e = (X_e^P \cup X_e^Q) \setminus X_i$;
- $I = I^P \times I^Q$;
- $T \subseteq Q_c \times Q_e \times Q_c$ *where* $(q, e, q') \in T$ *if* $(q[X_c^P], (e \cup q)[X_e^P], q'[X_c^P]) \in T^P$ *and* $(q[X_c^Q], (e \cup q)[X_e^Q], q'[X_c^Q]) \in T^Q$. ■

It follows from the definition that the transition relation of the composed process has the asynchronous property.

A *stuttering* step is a step in which the observation does not change. The interaction of a process with the environment is not influenced by the stuttering steps it takes. To capture this aspect, we first extend the notion of successor and take only observable (non-stuttering) moves into account. A *weak successor* can be obtained by a sequence of successors where all the steps are stuttering steps except the last step.

Definition 5. *Let $P = ((X_p, X_i, X_e), I, T)$ be a process, and q, q' states. We say q' is a* weak successor *of q, $q \longrightarrow_w^P q'$, if there are states $q_0 = q, q_1, \ldots, q_n = q' \in Q$ such that*

- *for all* $0 \leq i < n$. $q_i[X_o] = q_0[X_o]$; *and*
- *for all* $0 \leq i < n$. $q_i \longrightarrow^P q_{i+1}$; *and*
- $q_{n-1}[X_o] \neq q_n[X_o]$.

An execution *of P is a sequence $\bar{\sigma} = q_0 q_1 \cdots q_n$ in Q^*, where $q_0 \in I \times Q_e$ and $q_i \longrightarrow_w^P q_{i+1}$ for $0 \leq i < n$.* ■

The *trace* of an execution is its projection to observables:

Definition 6. *Let $P = ((X_p, X_i, X_e), I, T)$ be a process and $\bar{\sigma} = q_0 q_1 \cdots q_n$ an execution of P. The* trace *$tr(\bar{\sigma})$ of $\bar{\sigma}$ is a sequence in Q_o^*, defined to be $q_0[X_o] q_1[X_o] \cdots q_n[X_o]$. The* language *of a process P, $L(P)$, is defined to be the set of traces of all executions of P.* ■

Note that the language of a process completely determines its interaction with the environment: the language $P \| P'$ can be constructed from the languages $L(P)$ and $L(P')$.

We say process P *weakly refines* process Q if the projection of the trace of any execution of P is a trace of some execution of Q. Intuitively, P weakly refines Q if any observable behavior of P is also an observable behavior of Q modulo stuttering.

Definition 7. *Let* $P = ((X_p^P, X_i^P, X_e^P), I^P, T^P)$ *and* $Q = ((X_p^Q, X_i^Q, X_e^Q), I^Q, T^Q)$ *be two processes. Define* P weakly refines Q, $P < Q$, *if* $X_i^Q \subseteq X_i^P$, $X_o^Q \subseteq X_o^P$, *and* $\{\bar{\alpha}[X_o^Q] \ : \bar{\alpha} \in L(P)\} \subseteq L(Q)$. *We write* $P \cong Q$ *if both* $P < Q$ *and* $Q < P$. ∎

Note that the definition of the refinement allows the implementation to have more interface variables than the specification. The refinement relation defines a preorder over processes.

3 Refinement Checking

In the refinement verification problem, the user provides the process definitions for the implementation *Impl* and specification *Spec*, and our task is to check *Impl* $<$ *Spec*. The problem is computationally hard for two reasons:

1. Checking language inclusion requires determinizing the specification *Spec*. However, determinization requires subset construction, and may cause exponential blowup, and thus, should be avoided.
2. Since the definition of weak refinement considers stutter-closed traces, we need to consider the ϵ-closure of the specification *Spec* (that is, we need to match implementation transitions by the weak successors in specification). This problem is specific to the asynchronous setting. The computation of ϵ-closure is typically done by the transitive closure construction, and this is expensive.

3.1 Asynchronous Specification without Private Variables

If the specification has no private variables, all variables appear in the implementation as well. An observable implementation state corresponds to at most one state in the specification. Hence the first problem is resolved. In addition, since each specification transition is observable, the ϵ-closure of *Spec* is *Spec* itself. The refinement check then corresponds to verifying that (1) every initial state of *Impl* has a corresponding initial state of *Spec*, and (2) every reachable transition of *Impl* has a corresponding transition in *Spec*. This can be checked using the function shown in Figure 1. Notice that it is an on-the-fly algorithm and reports a violation once detected. It is easy to modify the algorithm to return, as a counter-example, the trace of *Impl* that is not a trace of *Spec*.

```
funct SimpleRefinement(s, Impl, Spec) ≡
  for t such that s ⟶^Impl t do
      if ¬(s[X_o^Spec] ⟶^Spec t[X_o^Spec]) then return false
      elsif t ∉ table then
           insert(table, t);
           if ¬SimpleRefinement(t, Impl, Spec) then return false
  od
  return true
```

Fig. 1. Algorithm for refinement checking

3.2 Specification with Private Variables

If the specification has private variables, the correspondence between implementation states and specification states should be provided by the user in order to make the checking feasible. The user needs to provide a module that assigns suitable values to the private variables of the specification in terms of values of implementation variables. This basic idea is needed even for manual proofs, and comes in various formalizations such as refinement maps [1], homomorphisms [17], forward-simulation maps [19], and witness modules [12, 21]. We formalize this using *witnesses.*

Definition 8. *Let* $Impl = ((X_p^I, X_i^I, X_e^I), I^I, T^I)$ *and* $Spec = ((X_p^S, X_i^S, X_e^S), I^S, T^S)$ *be two processes such that* $X_o^S \subseteq X_o^I$. *Then* $W = ((X_p^W, X_i^W, X_e^W), T^W)$ *is called a* witness *for Spec in Impl if* $X_i^W = X_p^S$, $X_e^W \subseteq X^I$ *and* $T^W \subseteq Q^W \times Q^W$. *We write* $q \longrightarrow^W q'$ *for* $(q, q') \in T^W$. ∎

Notice that the transition relation of a witness takes new values of its external variables into account. That is, witnesses are not asynchronous processes, and can proceed synchronously with the implementation.

Since our parallel composition was defined over processes, we need a new operation corresponding to product with the witness. We denote this operation by $\otimes$ (its formal definition is omitted due to lack of space). Once a proper witness is found, the refinement can be established by the following theorem.

Theorem 1. *Let Impl,* $Spec = ((X_p^S, X_i^S, X_e^S), I^S, T^S)$ *be processes and* W *be a witness for Spec in Impl. Define* $Spec^u = ((\emptyset, X_p^S \cup X_i^S, X_e^S), I^S, T^S)$. *Then* $Impl \otimes W \; < \; Spec^u$ *implies* $Impl \; < \; Spec$. ∎

If the verifier has expertise on the implementation, an efficient witness can be built based on this expert knowledge. Ideally, the witness should be *stateless*, and thus, should define the values of X_p^S as a *function* of the variables X^I. However, if the implementation is complicated, finding a proper witness may be difficult. In this case, one would like heuristics to fabricate the witness automatically.

Suppose the specification is composed of two subprocesses which control private and interface variables separately, say $Spec = Spec_p \| Spec_o$. The one controlling private variables ($Spec_p$) would be a candidate as a witness, for it updates

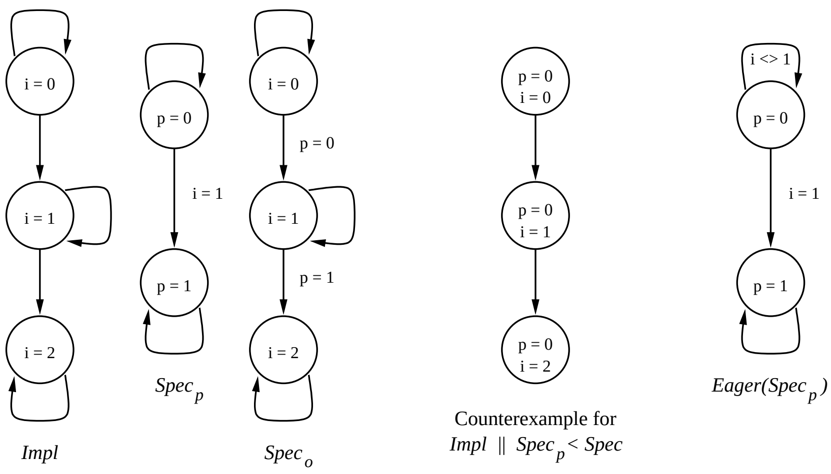


Fig. 2. Illustration of Eager Witness

the private variables of *Spec* based on the values of the observables of *Spec*, and hence of *Impl*. This is used in many of the case studies reported in [22, 11], and in fact, this choice of witness is complete if $Spec_p$ is deterministic. However, in our setting, $Spec_p$ is asynchronous, and is always nondeterministic as it may stutter at each step. In our examples, setting witness to $Spec_p$ does not work. Consider the example shown in Figure 2. Three processes *Impl*, $Spec_p$ and $Spec_o$ appear in the left. Suppose the variable p is private and i is an interface variable. It is easy to see that $Impl < Spec_p \| Spec_o$. If we were to use $Spec_p$ as a witness (assuming variables have their proper types), the middle figure gives a trace in $Impl \| Spec_p$ but not in $Spec_p \| Spec_o$. This problem results from the asynchronous property: private variables may not be updated because it is always possible to retain their old values regardless of observable variables. To avoid this, we would like them to react to the implementation updates immediately, whenever possible. This motivates the following definition:

Definition 9. *Let $P = ((X_p, X_i, X_e), I, T)$ be a process. Define the* eager variant *of P, $Eager(P)$, to be the process $((\emptyset, X_p \cup X_i, X_e), I, Eager(T))$ where $Eager(T) = T \setminus \{(q, e, q) :$ there exist $q' \neq q.(q, e, q') \in T\}$.* ■

In our modeling language, the eager variant of an asynchronous process can be constructed by a simple syntactic translation: we simply need to remove the keyword *lazy* from the atom declarations.

We propose to use $Eager(Spec_p)$ as a witness, and thus, the refinement check $Impl < Spec_p \| Spec_o$ is reduced to $Impl \otimes Eager(Spec_p) < Spec_p \| Spec_o$. The right figure in Figure 2 shows the synchronous witness $Eager(Spec_p)$, and it is easy to verify that $Impl \otimes Eager(Spec_p) < Spec_p \| Spec_o$ holds.

The assumption that *Spec* is decomposed into $Spec_p$ and $Spec_o$ can be relaxed if one can extract the private component $Spec_p$ from any given specification pro-

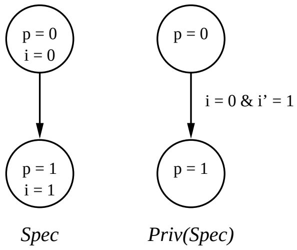


Fig. 3. Extraction of *Spec* Controlling Private Variables

cess *Spec* in a systematic way. Consider the transition shown in Figure 3 where i is an interface variable and p a private variable. We would like to construct a process *Priv(Spec)* that controls only p. Notice that variable i becomes an external variable in *Priv(Spec)*, and cannot be updated by it. Now the private variable p should be updated whenever i changes from 0 to 1. Since this extraction is used only to construct an eager witness, it can be synchronous, and can read new values of i. Hence we can construct a corresponding transition in *Priv(Spec)* as shown in the figure. This translation also can be automated easily in our modeling language by a simple syntactic transformation.

4 Hierarchic Reduction of Unobservable Transitions

Our reduction scheme is based on compressing sequences of transitions that neither read nor write observable variables. Such transitions are called *unobservable transitions* as formalized in the following definition.

Definition 10. *Let $P = (X, I, T)$ be a process. A transition $(q, e, q') \in T$ is* invisible *if*

- *it doesn't write to interface variables. $q[X_i] = q'[X_i]$; and*
- *it doesn't read from external variables. For all $e' \in Q_e$. $(q, e', q') \in T$.*

A transition is visible *if it is not invisible.* ■

It is worth emphasizing that invisibility is different from stuttering. It turns out that to check weak refinement, the number of invisible transitions taken by a process is not essential. Hence, we define a derivative of the process, which merges several invisible transitions together.

Definition 11. *Let $P = (X, I, T)$ be a process. The process* NEXT $P = (X, I, T')$ *satisfies the following condition: $(q, e, q') \in T'$ if $q = q'$ or there are states $q_0 = q, q_1, \ldots, q_n = q' \in Q_c$ such that $(q_i, e, q_{i+1}) \in T$ is invisible for $0 \leq i < n-1$; and $(q_{n-1}, e, q_n) \in T$ is visible.* ■

A useful property of the NEXT operator is that the weak refinement is a congruence relation with respect to it. In particular, NEXT $P \cong P$. Therefore, one may apply the NEXT operator to composite processes hierarchically. For example, instead of applying the NEXT of the composition of two processes, we can apply NEXT to the component processes and then compose them. In practice, the number of states of two NEXT processes composed together is less than the NEXT of the composition. This is due to the fact that NEXT can potentially reduce the number of interleavings of unobservable transitions of the components. Hence the number of intermediate states generated by composition is reduced, as will be illustrated in the examples of Section 6.

In [6], we had reported an on-the-fly algorithm to search such process expressions. A modification of that algorithm is used during refinement check. The algorithm is guaranteed to visit every reachable state of the given expression, and visits no more, and typically much less, than that required to explore the flat expression obtained by removing the applications of NEXT. The running time is linear in the number of states (more precisely, the transitions) visited by the algorithm, and there is basically no overhead in applying the reduction.

Since our modeling language distinguishes private, interface and external variables, our implementation can utilize this information to determine whether a transition is visible or not. In addition, the attributes of variables are syntactically determined, so the visibility of each transition can be recognized by parsing the model. Checking whether any transition is visible or not becomes a simple table lookup at runtime.

5 Compositional and Assume Guarantee Reasoning

A key property of the weak refinement relation is *compositionality*. It ensures that the refinement preorder is congruent with respect to the module operations.

Proposition 1. *(Compositionality) If* $P < Q$ *then* $P \| R < Q \| R$ *and* $\mathsf{hide}\ H\ \mathsf{in}\ P < \mathsf{hide}\ H\ \mathsf{in}\ Q$.

By applying the compositionality rule twice and using the transitivity of refinement it follows that, in order to prove that a complex compound module $P_1 \| P_2$ (with a large state space) implements a simpler compound module $Q_1 \| Q_2$ (with a small state space), it suffices to prove (1) P_1 implements Q_1 and (2) P_2 implements Q_2. We call this the *compositional proof rule*. It is valid, because parallel composition and refinement behave like language intersection and language containment, respectively.

While the compositional proof rule decomposes the verification task of proving implementation between compound modules into subtasks, it may not always be applicable. In particular, P_1 may not implement Q_1 for all environments, but only if the environment behaves like P_2, and vice versa. For example, consider a simple version of the *alternating bit protocol*.

Figure 4 shows the specification of the sender. The keywords *external, interface* and *private* classify the variables of a module as required in Definition 1. The transition relation of a module is given by several *atoms*, each atom having

```
type Pc is {snd, wait}
type message is {a, b, c}

module senderSpec is
  external ack : channel[1] of bool
  interface abp : channel[1] of bool; msg : channel[2] of messages;
            pcS : Pc; x, y : bool
  private m : message
  lazy atom sndSp
    controls send(abp), pcS, x, y, m
    reads receive(ack), pcS, x, m
  init
    [] true -> pcS' := snd; x' := false; y' := false; m' := nondet
  update
    [] pcS = snd -> send(abp, x); send(msg, m); pcS' := wait
    [] pcS = wait & ¬isEmpty(ack) ->
          receive(ack, y); x' := ¬x; m' := nondet; pcS' := snd
```

Fig. 4. Specification of Sender Process

an exclusive update right on a subset of the controlled variables. In the asynchronous case, atoms can be viewed as (atomic) processes. The way an atom initializes and updates its controlled variables is given by a *guarded command.* The primed notation stands for new values of variables. The *reads* and *controls* keywords allow to define the variables read and written by an atom. The *lazy* keyword corresponds to our asynchronous condition and allows stuttering.

To simplify the asynchronous specification style, we support a predefined *channel type.* This is essentially a record type, consisting of a ring buffer, a sender and a receiver pointer. Note however, that in our setting, the fields of this record may be controlled by different modules (or atoms) and this has to be made explicit. The sender specification simply assumes that the sender and the receiver are synchronized with each other by using the boolean one element channels `abp` and `ack`. The implementation of the sender (Figure 5) makes sure that the sender works properly in an environment that may lose messages. Each time the sender sends a new message along the channel `msg` it toggles the bit `x` and sends it along the channel `abp`. If it does not receive this bit back along the channel `ack` it resends the message. The receiver module works in a symmetric way.

Trying to prove now that `senderImp` $<$ `senderSpec` or that `receiverImp` $<$ `receiverSpec` fails. The implementation of the sender and receiver refine their abstract counterparts only in an environment that behaves like the abstract receiver and sender respectively. For such cases, an *assume-guarantee* proof rule is needed [27, 10, 3, 4]. Our rule differs from the earlier ones in that it uses a different notion of refinement, namely, the stutter-closed one.

Proposition 2. *(Assume-Guarantee) If* $P_1 \| Q_2 < Q_1 \| Q_2$ *and* $Q_1 \| P_2 < Q_1 \| Q_2$*, then* $P_1 \| P_2 < Q_1 \| Q_2$.

```
module senderImp is
  external ack : channel[2] of bool
  interface abp : channel[2] of bool; msg : channel[2] of message;
            pcS : Pc; x, y : bool
  private m : message
  lazy atom sndSp
    controls send(abp), pcS, x, y, m
    reads receive(ack), pcS, x, y
  init
      [] true -> pcS' := snd; x' := false; y' := false; m':= nondet
  update
      [] pcS = snd -> send(abp, x); send(msg, m); pcS' := wait
      [] pcS = wait & first(ack, ¬x) -> receive(ack, y); pcS' := snd
      [] pcS = wait & first(ack, x) ->
            receive(ack, y); x' := ¬x; m':= nondet; pcS' := snd
```

Fig. 5. Implementation of Sender Process

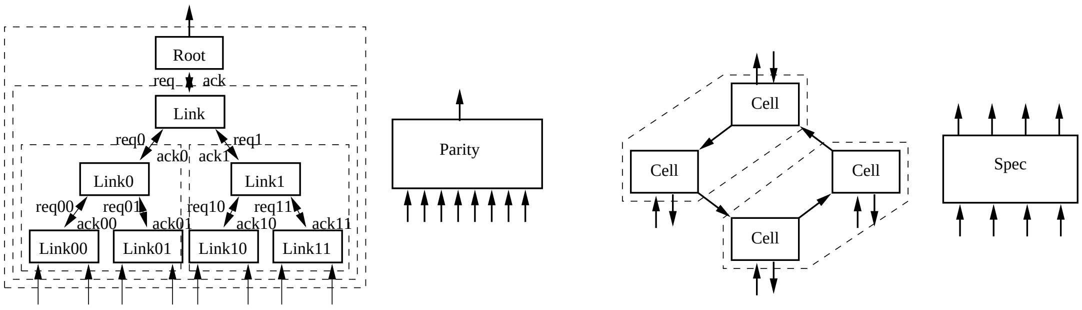
Since $P_1 \| P_2$ has the largest state space, both proof obligations typically involve smaller state spaces than the original proof obligation. The assume-guarantee proof rule is circular; unlike the compositional proof rule, it does not simply follow from the fact that parallel composition and implementation behave like language intersection and language containment. Rather the proof of the validity of the assume-guarantee proof rule proceeds by induction on the length of traces. For this, it is crucial that every trace of a module can be extended. An alternative version of the rule states that "if $P_1 \| Q_2 < Q_1$ and $Q_1 \| P_2 < Q_2$, then $P_1 \| P_2 < Q_1 \| Q_2$." For the enumerative checker, we prefer the variant stated in Proposition 2 because it constrains the right hand side and therefore reduces the required computation.

The soundness of assume-guarantee depends crucially on the asynchronous property of the processes (it is not valid for arbitrary reactive modules). Recall that witnesses are not processes. This implies that witnesses can be introduced only after generating all subgoals using assume-guarantee since the assume guarantee requires asynchronicity.

6 Case Studies

6.1 Tree-structured parity computer

We will demonstrate our proposed techniques in a tree-structured process that computes a function, say, *parity*, of the requests received from the leaf clients, where each request supplies value that is either 0 or 1 (Figure 6). Each client sends requests and receives acknowledgements to and from its parent. Each link process gathers requests from its children, computes the parity, and reports it to its parent. The root process calculates the final result and sends the result to the children link processes, which in turn, propagate the result down.



Parity Computer Leader Election

Fig. 6. Parity Computer and Leader Election

Let us focus on the process *System* described as

$$\text{Root} \parallel \text{Link} \parallel \text{Link0} \parallel \text{Link1} \parallel \text{Link00} \parallel \text{Link01} \parallel \text{Link10} \parallel \text{Link11}$$

We can naturally cluster subprocesses as shown in Figure 6, replace the process *System* by the process *NextSystem*:

$$\text{NEXT hide} \left[\, \text{Root} \parallel \text{NEXT hide} \left[\, \text{Link} \parallel \begin{matrix} \text{NEXT hide } (\text{Link0} \parallel \text{Link00} \parallel \text{Link01}) \\ \text{NEXT hide } (\text{Link1} \parallel \text{Link10} \parallel \text{Link11}) \end{matrix} \right]\right].$$

For readability the argument variables to **hide** are not shown above, but should be obvious from context: only the variables communicating with the parent need to be observable at each level of the subtree. Note that a step of the tree rooted at *Link* is composed of a step of the process *Link*, and a sequence of steps of the subtree rooted at *Link0* until *req0* or *ack0* is accessed, and a sequence of steps of the subtree rooted at *Link1* until *req1* or *ack1* is accessed. Since each node communicates with parent only after it has received and processed requests from its children, we get excellent reduction. Furthermore, the congruence property ensures that *System* can be replaced by *NextSystem* during any refinement check.

The specification *Parity* is an n-way process that simply calculates the parity based on all of its inputs. However, since the input signals may not arrive at the same time, *Parity* needs private variables to record processed signals. Our goal is to check if *System* $<$ *Parity* holds. In this case, the correspondence between the private variables of the two descriptions is unclear. To construct a witness, the first step is to extract the process *Priv(Parity)* from *Parity*, and then, use its eager version as the witness. This automatic construction turns out to be adequate in this case. Then we apply the hierarchic reduction and check $\mathit{NextSystem} \| \mathit{Eager}(\mathit{Priv(Parity)}) < \mathit{Parity}$. Figure 7 shows the number of states stored in the table when we check the refinement. As indicated, our hierarchical reduction is quite effective in alleviating the state-explosion problem.

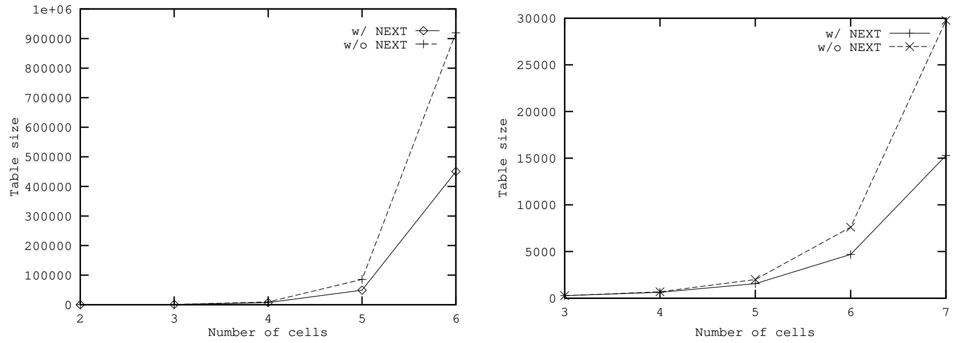


Fig. 7. Table Size of Parity Computer and Leader Election Examples

6.2 Leader election

The leader election problem consists of cells, each with a unique initial number, connected in a ring (see Figure 6). The problem is to elect a leader, which is achieved by determining who has the greatest number. Each cell can have one of three statuses: `unknown`, `chosen`, and `not_chosen`. Initially, all cells have status `unknown`. In the end, only the cell with the greatest number should have status `chosen`, and all other cells should have status `not_chosen`. Variants of this problem have been extensively used to illustrate temporal logic model checking. We show consistency of a distributed leader election protocol with respect to the process *Spec* that determines all statuses in a centralized manner by comparing the given numbers. Since there is no private variable in *Spec*, we can directly check if $System < Spec$. The result is shown in Figure 7, which indicates effectiveness of the hierarchical reduction (see Figure 6 for clustering for the hierarchical reduction).

6.3 Distributed mutual exclusion

In Distributed Mutual Exclusion (DME) [20], a ring of cells are connected to a user process each. Figure 8 shows the specification of a cell.

In the specification, a virtual token is passed around the cells clockwise. Initially, only one cell has the token. Each cell can send requests to its right neighbor (`right_req`) and acknowledgements to its left neighbor and user (`left_ack` and `user_ack`). When it receives a request from its left neighbor, it checks if it possesses the token. If it does not, a request is sent to its right neighbor. Otherwise, it passes the token by setting `left_ack` to true. After its left neighbor receives the token, it resets left acknowledgement to false.

It handles user's requests similarly. However, when the user releases the token by assigning `user_req` to false, the cell resets not only the user acknowledgement to false, but also the token variable to true to obtain the token from the user. Finally, if it sends request to its right neighbor and the neighbor passes the token, it sets the token variable to true and the right acknowledgement to false.

```
module CSpec_T is
  external left_req, right_ack, user_req : bool
  interface left_ack, right_req, user_ack : bool; token : bool
  lazy atom CSPEC_T
    controls left_ack, right_req, user_ack, token
    reads left_req, left_ack, right_req, right_ack,
          user_req, user_ack, token
  init
    [] true -> left_ack' := false; right_req' := false;
               user_ack' := false; token' := true
  update
    [] left_req & ¬left_ack & ¬right_ack & ¬token -> right_req' := true
    [] left_req & ¬left_ack & token -> left_ack' := true; token' := false
    [] left_ack & ¬left_req -> left_ack' := false
    [] user_req & ¬user_ack & ¬right_ack & ¬token -> right_req' := true
    [] user_req & ¬user_ack & token -> user_ack' := true; token' := false
    [] user_ack & ¬user_req -> user_ack' := false; token' := true
    [] right_req & right_ack -> token' := true; right_req' := false
```

Fig. 8. Specification of a DME Cell with Token

Each cell is implemented by basic logic gates (10 ANDs, 2 ORs, 7 NOTs) and special components (C-element and 2-way Mutual Exclusion blocks). The user process simply sends requests and receives acknowledgements accordingly. Previous verification studies of DME implementation have considered checking requirements written in a temporal logic such as CTL. We define the process *System* to be the composition of three cells and user processes. The specification process *Spec*, then, is the high-level distributed algorithm shown in figure 8. Our goal is to check whether *System* $<$ *Spec*.

In order to make checking *System* $<$ *Spec* feasible, we would like to avoid non-deterministic choices of variable values in *Spec*. Hence, we need to establish the relation among variables of *System* and of *Spec*. Notice that the high level specification *Spec* uses the virtual token. Because the implementation *System* consists of circuit signals, it is rather difficult, if not impossible, to establish the desired relation. We therefore apply the technique in section 3.2 to extract the witness module for each cell.

In figure 9, module `Wit_T` is the witness module *Eager*(*Priv(Cell_T)*). It is easy to see that the witness can be generated by syntactic transformation (cf. figure 8). We thus define the witness module *Eager*(*Priv(Spec)*) to be the composition of cell witnesses and check *System* $\otimes$ *Eager*(*Priv(Spec)*) $<$ *Spec*. We can apply hierarchical reduction on the refinement checking. Alternatively, we may apply assume guarantee proof rule first, obtain the following three proof obligations, and then generate witnesses for each obligations.

Figure 10 shows the table size of each experiments. We can see that Next operator reduces the number of states while checking the whole system. There are three obligations in assume guarantee proof rule for *System* consists of three

```
module Wit_T is
  external left_req, right_ack, user_req : bool;
           left_ack, right_req, user_ack : bool
  interface token : bool
  atom WIT_T
    controls token
    reads left_req, right_req, right_ack, user_req, user_ack, token
    awaits left_ack, user_ack, right_req
  init
    [] true -> token' := true
  update
    [] left_req & ¬left_ack & token & left_ack' -> token' := false
    [] user_req & ¬user_ack & token & user_ack' -> token' := false
    [] user_ack & ¬user_req & ¬user_ack' -> token' := true
    [] right_req & right_ack & ¬right_req' -> token' := true
```

Fig. 9. Witness Module for DME Cell

	with Next		without Next
without assume guarantee	4128		6579
with assume guarantee	$Cell_0 \| Spec_1 \| Spec_2 < Spec_0$	3088	3088
	$Spec_0 \| Cell_1 \| Spec_2 < Spec_1$	3088	
	$Spec_0 \| Spec_1 \| Cell_2 < Spec_2$	3088	

Fig. 10. Table Size of Distributed Mutual Exclusion

cells. They all use the same number of states, which is less than checking the whole system. However, Next operator does not save space for obligations.

7 Conclusions

We have proposed a methodology, with an associated tool, for checking refinement based on stutter-closed language inclusion. Our notion of refinement is the natural one for asynchronous processes, and in many cases, writing the specification as a process is more intuitive than writing a set of temporal logic assertions. Let us summarize our approach to refinement checking.

1. The input refinement problem concerning weak refinement of asynchronous descriptions is decomposed into subgoals using compositional and assume-guarantee rules.
2. For each subgoal of the form $Impl < Spec$ is replaced by the check $Impl \otimes W < Spec$, where W is a (synchronous) witness. The witness may either be supplied by the user, or automatically chosen to be $Eager(Priv(Spec))$.
3. Checking $Impl \otimes W < Spec$ corresponds to a reachability analysis. During the search, *Impl* is optimized by replacing each subexpression E in *Impl* by Next E.

We have reported on an enumerative refinement checker and a proof assistant. Note that the first and the third items above exploit the hierarchical constructs in our language. The methodology is illustrated based on traditional benchmarks involving asynchronous processes. We conclude by summarizing our experience about different heuristics.

Assume-guarantee reasoning. Its application requires decomposing the specification into components, and this can be tedious. The subgoals can be of much lower complexity than the original goals. However, particularly in the context of enumerative algorithms, subgoals can also be more difficult to check since specifications tend to be nondeterministic or partial. Thus, effectiveness of this technique has been mixed.

Automatic witness construction. We have proposed a simple heuristic for constructing a synchronous witness from the specification. This approach has been surprisingly effective in our examples.

Hierarchical reduction. The algorithm to compress invisible transitions in a recursive manner is easy to implement with no noticeable overhead, and good reduction in many cases. This method is incomparable to and compatible with symmetry reduction [8, 16]. When compared to the partial-order reduction method [24, 9], for the problem of invariant verification our method gives less reduction at least in the leader election example. However, it is unclear if partial-order reduction is applicable for refinement checking in presence of synchronous witnesses.

Acknowledgements

We thank all the members of the MOCHA team at University of California at Berkeley and at University of Pennsylvania for their assistance. This research was supported in part by NSF CAREER award CCR97-34115, by DARPA/NASA grant NAG2-1214, by SRC contract 99-TJ-688, by Bell Laboratories, Lucent Technologies, and by Alfred P. Sloan Faculty Fellowship.

References

1. M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, 1991.
2. M. Abadi and L. Lamport. Composing specifications. *ACM TOPLAS*, 15(1):73–132, 1993.
3. M. Abadi and L. Lamport. Conjoining specifications. *ACM TOPLAS*, 17:507–534, 1995.
4. R. Alur and T. Henzinger. Reactive modules. *Formal Methods in System Design*, 15(1):7–48, 1999.
5. R. Alur, T. Henzinger, F. Mang, S. Qadeer, S. Rajamani, and S. Tasiran. MOCHA: Modularity in model checking. In *CAV'98: Computer Aided Verification*, LNCS 1427, pp. 516–520, 1998.

6. R. Alur and B.-Y. Wang. "Next" heuristic for on-the-fly model checking. In *CONCUR'99: Concurrency Theory, Tenth International Conference*, LNCS 1664, pp. 98–113, 1999.
7. R. Cleaveland, J. Parrow, and B. Steffen. The concurrency workbench: A semantics-based tool for the verification of finite-state systems. *ACM Trans. on Programming Languages and Systems*, 15(1):36–72, 1993.
8. E. Emerson and A. Sistla. Symmetry and model checking. In *CAV'93: Computer-Aided Verification*, LNCS 697, pp. 463–478, 1993.
9. P. Godefroid. Using partial orders to improve automatic verification methods. In E. Clarke and R. Kurshan, editors, *CAV'90: Computer-Aided Verification*, LNCS 531, pp. 176–185, 1990.
10. O. Grümberg and D. Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems*, 16(3):843–871, 1994.
11. T. Henzinger, X. Liu, S. Qadeer, and S. Rajamani. Formal specification and verification of a dataflow processor array. In *ICCAD'99: International Conference on Computer-aided Design*, pp. 494–499, 1999.
12. T. Henzinger, S. Qadeer, and S. Rajamani. You assume, we guarantee: Methodology and case studies. In *CAV'98: Computer-aided Verification*, LNCS 1427, pp. 521–525, 1998.
13. T. Henzinger, S. Qadeer, and S. Rajamani. Assume-guarantee refinement between different time scales. In *CAV'99: Computer-aided Verification*, LNCS 1633, pp. 208–221, 1999.
14. C. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.
15. G. Holzmann. The model checker SPIN. *IEEE Trans. on Software Engineering*, 23(5):279–295, 1997.
16. C. Ip and D. Dill. Verifying systems with replicated components in murφ. In *Computer Aided Verification*, LNCS 1102, 1996.
17. R. Kurshan. *Computer-aided Verification of Coordinating Processes: the automata-theoretic approach*. Princeton University Press, 1994.
18. N. Lynch. *Distributed algorithms*. Morgan Kaufmann, 1996.
19. N. Lynch and M. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the Seventh ACM Symposium on Principles of Distributed Computing*, pp. 137–151, 1987.
20. A. Martin. The design of a self-timed circuit for distributed mutual exclusion. In *Chapel Hill Conference on Very Large Scale Integration*, pp. 245–260, 1985.
21. K. McMillan. A compositional rule for hardware design refinement. In *Computer-Aided Verification*, LNCS 1254, pp. 24–35, 1997.
22. K. McMillan. Verification of an implementation of tomasulo's algorithm by compositional model checking. In *CAV'98: Computer-Aided Verification*, LNCS 1427, pp. 110–121, 1998.
23. R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.
24. D. Peled. Combining partial order reductions with on-the-fly model-checking. In *CAV'94: Computer Aided Verification*, LNCS 818, 1994.
25. A. Pnueli. Applications of temporal logic to the specification and verification of reactive systems: a survey of current trends. In *Current Trends in Concurrency*, LNCS 224, pp. 510–584, 1986.
26. J. Scattergood. *The semantics and implementation of machine-readable CSP*. PhD thesis, Oxford University, 1998.
27. E. Stark. A proof technique for rely-guarantee properties. In *Foundations of Software Technology and Theoretical Computer Science*, LNCS 206, pp. 369–391, 1985.