Verifying Network Protocol Implementations by Symbolic Refinement Checking

Rajeev Alur and Bow-Yaw Wang

Department of Computer and Information Science
University of Pennsylvania
`[alur,bywang] @ cis.upenn.edu`
`www.cis.upenn.edu/~[alur,bywang]`

Abstract. We consider the problem of establishing consistency of code implementing a network protocol with respect to the documentation as a standard RFC. The problem is formulated as a refinement checking between two models, the implementation extracted from code and the specification extracted from RFC. After simplifications based on assume-guarantee reasoning, and automatic construction of witness modules to deal with the hidden specification state, the refinement checking problem reduces to checking transition invariants. The methodology is illustrated on two case-studies involving popular network protocols, namely, PPP (point-to-point protocol for establishing connections remotely) and DHCP (dynamic-host-configuration-protocol for configuration management in mobile networks). We also present a symbolic implementation of a reduction scheme based on compressing internal transitions in a hierarchical manner, and demonstrate the resulting savings for refinement checking in terms of memory size.

1 Introduction

Network protocols have been a popular domain of application for model checkers for over a decade (see, for instance, [15, 10]). A typical application involves checking temporal requirements, such as absence of deadlocks and eventual transmission, of a model of a network protocol, such as TCP, extracted from a textbook description or a standard documentation such as a network RFC (Request for Comments) document. While this approach is effective in detecting logical errors in a protocol design, there is still a need to formally analyze the actual implementation of the protocol standard to reveal implementation errors. While analyzing the code implementing a protocol, the standard specification, typically available as a network RFC, can be viewed as the abstract model. Since the standard provides implementation guidelines for different vendors on different platforms, analysis tools to detect inconsistencies with respect to the standard can greatly enhance the benefits of standardization.

The problem of verifying a protocol implementation with respect to its standardized documentation can naturally be formulated as *refinement checking*. The implementation model I is extracted from the code and the specification model S is extracted from the RFC document. We wish to verify that $I \preceq S$ holds, where the notion $\preceq$ of refinement is based on language inclusion. A recent promising approach to *automated* refinement checking combines assume-guarantee reasoning with search algorithms [19, 14, 4], and has been successfully applied to synchronous hardware designs such as pipelined processors [20] and a VGI chip [13].

To establish the refinement, we employ the following three-step methodology (advocated, for instance in [4]). First, the refinement obligation is used to generate simpler subgoals by applying *assume guarantee reasoning* [23, 2, 5, 12, 19]. This 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. Second concerns verification of a subgoal $I \preceq S$, when S 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 disguises such as refinement maps [1], homomorphisms [17], forward-simulation maps [18], and witness modules [14, 19]). Consequently, the refinement check $I \preceq S$ reduces to $I \| W \preceq S$, where W is the user-supplied witness for private variables of S. As a heuristic for choosing W automatically, we had proposed a simple construction that transforms S to $\mathit{Eager}(S)$, which is like S, but takes a stuttering step only when all other choices are disabled [4]. Once a proper witness is chosen, the third and final step requires establishing that every reachable transition of the implementation has a matching transition of the specification, and can be done by an algorithmic state-space analysis for checking transition invariants.

For performing the reachability analysis required for verifying transition invariants efficiently, we propose an optimization of the symbolic search. The proposed algorithm is an adaptation of a corresponding enumerative scheme based on compressing unobservable transitions in a hierarchical manner [6]. The basic idea is to describe the implementation I in a hierarchical manner so that I is a tree whose leaves are atomic processes, and internal nodes compose their children and hide as many variables as possible. This suggests a natural optimization: while computing the successors of a state corresponding to the execution of a process, apply the transition relation repeatedly until a shared variable is accessed. A more effective strategy is to apply the reduction in a recursive manner exploiting the hierarchical structure. In this paper, we show how this hierarchical scheme can be implemented symbolically, and establish significant reductions in space and time requirements.

Our methodology for refinement checking is implemented in the model checker MOCHA [3]. Our first case study involves verifying part of the RFC specification of Point-to-Point Protocol (PPP) widely used to transmit multi-protocol datagrams [22]. The implementation `ppp` version 2.4.0 is an open-source package included in various Linux distributions. We extract the model `ppp` of the specification and the model `pppd` of the implementation manually. To establish the refinement, we need to assume that the communication partner behaves like the specification model, thus, employ assume-guarantee reasoning. The specification has many private variables, and we use the "eager witness" construction to reduce the problem to transition invariant check. Our analysis reveals an inconsistency between the C-code and the RFC document. The second case study concerns the Dynamic Host Configuration Protocol (DHCP) that provides a standard mechanism to obtain configuration parameters. We analyze the `dhcp` package version 2.0 patch level 5, the standard implementation distributed by Internet Software Consortium, with respect to its specification RFC 2131 [11].

2 Refinement Checking

In this section, we summarize the definition of processes, refinement relation over processes, and the methodology for refinement checking. The details can be found in [4].

The process model is a special class of *reactive modules* [5] that corresponds to asynchronous processes communicating via read-shared write-exclusive variables. A process is defined by the set of its variables, along with the constraints for initializing and updating variables. The variables of a process P are partitioned into three classes: *private* variables that cannot be read nor 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 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 private and observable variables is essential to applying our optimization algorithm based on compressing internal transitions. The state space of the process is the set of possible valuations to all its variables. A state is also partitioned into different components as the variables are, for instance, controlled state and external state. The initial predicate specifies initial controlled states, and the transition predicate specifies how the controlled state is changed according to the current state.

In the following discussion, we write $\mathcal{B}[X]$ for the set of predicates over variables in X. For the set of variables X, we write X' for the corresponding variables denoting updated values after executing a transition. Furthermore, for sets of variables $X = \{x_i\}$ and $Y = \{y_i\}$ with the same cardinality, $X = Y$ denotes $\wedge_i x_i = y_i$. For any subset Z of variables X and $P \in \mathcal{B}[X]$, $\exists Z.P$ and $\forall Z.P$ stand for the existential and universal quantification over the variables in Z.

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. We define* $X_c = X_p \cup X_i$ *to be the* controlled variables, *and* $X_o = X_i \cup X_e$ *to be the* observable variables*;*
- *Given a set* X *of typed variables, a* state *over* X *is an assignment of variables to their values. We define* Q_c *to be the set of* controlled states *over* X_c, Q_e

to be the set of external states *over* X_e*,* $Q = Q_c \times Q_e$ *to be the set of states, and* Q_o *to be the set of* observable states *over* X_o*;*
- $I \in \mathcal{B}[X_c]$ *is the* initial predicate*;*
- $T \in \mathcal{B}[X, X'_c]$ *is the* transition predicate *with the property (called* asynchronous property*) that* $(X'_c = X_c) \Rightarrow T$*.*

■

The asynchronous property says that a process may idle at any step, and thus, the speeds of the process and its environment are independent. In order to support structured descriptions, we would like to build complex processes from simple ones. Three constructs, $\mathsf{hide}\ H\ \mathsf{in}\ P$, $P \| P'$ and $P[X := Y]$ for building new processes are defined. The hiding operator makes interface variables inaccessible to other processes, and its judicious use allows more transitions to be considered internal. 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. The transition predicate of $P \| Q$ is thus the conjunction of transition predicates of P and Q. The renaming operator $P[X := Y]$ substitutes variables X in P by Y.

For a process P, the sets of its executions and observable traces are defined in the standard way. Given two processes P and Q, we say P *refines* Q, written $P \preceq Q$, if each observable trace of P is an observable trace of Q. Checking refinement relation is computationally hard, and we simplify the problem in two ways. First, our notion of refinement supports an *assume guarantee* principle which asserts that it suffices to establish separately $P_1 \| Q_2 \preceq Q_1$ and $Q_1 \| P_2 \preceq Q_2$ in order to prove $P_1 \| P_2 \preceq Q_1 \| Q_2$. This principle, similar in spirit to many previous proposals [23, 2, 5, 12, 19], is used to reduce 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. The second technique reduces checking language inclusion to verifying transition invariants. If the specification has no private variables, an observable implementation state corresponds to at most one state in the specification. The refinement check then corresponds to verifying that every initial state of P has a corresponding initial state of Q, and every reachable transition of P has a corresponding transition in Q. When Q has private variables, then 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 witness W that assigns suitable values to the private variables of the specification in terms of implementation variables. It can be shown that $P \preceq Q$ follows from establishing $P \| W \preceq Q$. In our setting of asynchronous processes, it turns out that the witness W itself should not be asynchronous (that is, for asynchronous W, $P \| W \preceq Q$ typically does not hold). This implies that the standard trick of choosing the witness to be the subprocess Q^p of Q that updates its private variables, used in many of the case studies reported in [20, 13], does not work in the asynchronous setting. As a heuristic for choosing W automatically, we have proposed a construction that transforms Q^p to *Eager*(Q^p), which is similar to the subprocess Q^p, but takes a stuttering step only when all other choices are disabled [4]. 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 proportional to the product of P and Q^p.

3 Symbolic Search with Hierarchical Reduction

In this section, we consider the problem of verifying $P \preceq Q$ when Q does not have any private variables. In this case, if one can check that all reachable P transitions have corresponding transitions in Q, then $P \preceq Q$ holds. Since all variables of Q appear in P, the corresponding transitions can be obtained by projection, and the problem can be solved by an appropriately modified reachability analysis. The core routine is *Next*: given a process P and a set R of its states, $Next(P, R)$ returns the set T of transitions of P starting in R along with the set S of successors of R. There is, however, a practical problem if one intends to implement the successor function *Next* with existing BDD packages. Since *Next* needs to return the set of transitions, early quantification, an essential technique for image computation, is less effective. In [6], we have reported a heuristic to improve the enumerative search algorithm. In this section, we propose a symbolic algorithm to implement it.

We use NEXT P represent the process obtained by merging "invisible" transitions of P where invisibility is defined to be both write-invisible (not writing to interface variables) and read-invisible (not reading from external variables). Let $T \in \mathcal{B}[X_p, X_i, X_o, X'_p, X'_i]$ be a transition predicate (the primed variables denote the updated values). The write-invisible transitions are captured by the predicate $T \wedge (X_i = X'_i)$ (the second clause says that the interface variables stay unchanged) and read-invisible transitions correspond to $\forall X_e.T$ (the quantification ensures that the transition is not dependent on external variables). Thus, the *invisible* component T_i of T is $T \wedge (X_i = X'_i) \wedge \forall X_e.T$, and the visible component T_v is $T \wedge \neg T_i$. Define the concatenation $T_1 \bowtie T_2$ of two transition predicates $T_1, T_2 \in \mathcal{B}[X, X']$ to be $\exists Z. T_1[X' \leftarrow Z] \wedge T_2[X \leftarrow Z]$.

Definition 2. *Let* $P = ((X_p, X_i, X_o), I, T)$ *be a process. Define* NEXT $P = ((X_p, X_i, X_o), I, T')$ *with* $T' = (X_c = X'_c) \vee (\mu S. T_v \vee (T_i \bowtie S))$. ■

The transition predicate of NEXT P is equivalent to $(X = X') \vee T_v^P \vee (T_i^P \bowtie T_v^P) \vee (T_i^P \bowtie T_i^P \bowtie T_v^P) \vee \cdots$. In other words, a transition in NEXT P is either a stuttering transition, or zero or more invisible transitions followed by a visible transition of P.

It can be shown that NEXT P and P are equivalent (modulo stuttering). Furthermore, the NEXT operator is congruent [4]. This allows us to apply the NEXT operator to every subprocess of a process constructed by parallel composition, hiding and instantiation. We proceed to describe a symbolic algorithm for state-space analysis of a process expression with nested applications of NEXT , without precomputing the transition relations of the subprocesses (such a precomputation would require an expensive transitive closure computation).

$$
\begin{array}{l}
\underline{\textbf{funct}}\ Next(M, R)\ \equiv \\
\quad \underline{\textbf{if}}\ M \equiv P \\
\quad\quad \underline{\textbf{then}}\ helper := \lambda Q.\underline{\textbf{let}}\ Q_c := Q[X_c^P] \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad T_c := T_P \wedge Q_c \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad R' := (\exists X_c^P . T_P \wedge Q)[X_c'^P \leftarrow X_c^P] \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad R'' := R' \setminus cache \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad cache := cache \vee R' \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad \underline{\textbf{in}}\ (T_c, R'') \\
\quad\quad\quad\quad \underline{\textbf{return}}\ NextAuc(P, helper, R) \\
\quad \underline{\textbf{elsif}}\ M \equiv M_1 \| M_2 \\
\quad\quad\quad \underline{\textbf{then}}\ (T_1, N_1) := Next(M_1, R) \\
\quad\quad\quad\quad\quad (T_2, N_2) := Next(M_2, R) \\
\quad\quad\quad\quad\quad S_1 = (\exists X_c^{M_2}.R) \wedge (X_c'^{M_1} = X_c^{M_1}) \\
\quad\quad\quad\quad\quad S_2 = (\exists X_c^{M_1}.R) \wedge (X_c'^{M_2} = X_c^{M_2}) \\
\quad\quad\quad\quad\quad T := (T_1 \wedge S_2) \vee (T_2 \wedge S_1) \vee (T_1 \wedge T_2) \\
\quad\quad\quad\quad\quad N' := (\exists X_c^M . R \wedge T_1 \wedge T_2)[X_c'^M \leftarrow X_c^M] \\
\quad\quad\quad\quad\quad N := N_1 \vee N_2 \vee N' \\
\quad\quad\quad\quad\quad \underline{\textbf{return}}\ (T, N) \\
\quad \underline{\textbf{elsif}}\ M \equiv hide Y in M_1 \\
\quad\quad\quad \underline{\textbf{then}}\ helper := \lambda Q.Next(M_1, Q) \\
\quad\quad\quad\quad\quad \underline{\textbf{return}}\ NextAuc(M, helper, R) \\
\quad \underline{\textbf{fi}}
\end{array}
$$

Fig. 1. Algorithm *Next*

The algorithm *Next* (figure 1) computes the visible transitions of a process M from the current states R by proceeding according to the structure of M. For each case, a tuple of transitions and a set of new states is returned. Each atomic process takes its turn to update its controlled variables as the algorithm traverses the expression. Whenever a state is reached by the current exploration, we check if it has been visited. If not, the state is put in the newly reached states. The transition compression of subprocesses is performed by applying NEXT implicitly in cases of atomic processes and hiding. This is achieved by invoking the function *NextAuc* to merge invisible transitions in these two cases. For parallel composition $M_1 \| M_2$, it is not necessary to do so since variable visibility remains the same. Therefore, the algorithm simply invokes itself recursively to obtain transitions T_1 and T_2 corresponding to subprocesses M_1 and M_2 respectively, and computes the composed transitions for the following three cases: (1) M_1 takes a transition in T_1 and M_2 stutters; (2) M_2 takes a transition in T_2 and M_1 stutters; and (3) both M_1 and M_2 take transitions in T_1 and T_2 respectively.

For atomic processes and the case of hiding, the *helper* function is given to *NextAuc* as a parameter. It returns transitions and new states of the subprocess before NEXT is applied. For hiding, the *helper* function simply returns the transitions and new states of M_1, and the algorithm *Next* lets *NextAuc* do the transition compression. For an atomic process, the *helper* function computes

comment: *helper* returns a tuple of lower-level transitions
comment: and newly reached states from the given set of states.
funct $NextAuc(M, helper, R) \equiv$
$N := false$
$T := false$
$I := true$
$Q := R$
do
$(T', N') := helper(Q)$
$T_i' := (T' \wedge (X_c'^M = X_c^M)) \wedge (\forall X_e^M.T')$
$T := (I \bowtie T') \vee T$
$I := I \bowtie T_i'$
$Q' := (\exists X_c^M.Q \wedge T_i')[X_c'^M \leftarrow X_c^M]$
$Q := N' \wedge Q'$
$N := N \vee (N' \setminus Q)$
while $Q \neq \emptyset$
return (T, N)

Fig. 2. Algorithm *NextAuc*

transitions T_c and new states R''. It then returns the transitions and newly reached states after updating cache.

Figure 2 shows the *NextAuc* algorithm for invisible transition compression. The naive fixed-point computation hinted in definition 2 is expensive and unnecessary. Rather than computing fixed points, our algorithm generates the transition predicate of NEXT P on the fly by considering only the current states. The idea is to compute $T_i \bowtie \cdots \bowtie T_i \bowtie T_v$ incrementally until all visible transitions reachable from the current states are generated. Several variables are kept by the algorithm to perform the task. N accumulates newly reached states in each iteration, T consists of compressed transitions, I is the concatenation of consecutive invisible transitions and Q is the states reached by invisible transitions in the current iteration.

The algorithm *NextAuc* first computes the invisible component T_i' in T'. The new transitions T' are added to T after concatenated with previous invisible transitions. The concatenated invisible transition I is updated by appending T_i'. To compute states for the next iteration, the set Q' of all reached states by current invisible transitions is generated. The new states Q reached by invisible transitions are the intersection of the newly reached states N' and invisible states Q'. Finally, the visible states of N' are put into the new visible states N. The main correctness argument about the algorithm is summarized by:

Theorem 1. *Let M be a process, $R \in \mathcal{B}[X^M]$ and suppose Next(M, R) returns (T, N). Then the predicate $T \wedge R$ captures the transitions of* NEXT M *starting in R, and N contains all successor states of R that are not previously visited.* ∎

Implementation. The symbolic algorithm for refinement checking is implemented in the model checker MOCHA [3]. The implementation is in Java using

Event	Action
Up : lower layer is Up	tlu : This-Layer-Up
Down : lower layer is Down	tld : This-Layer-Down
Open : administrative Open	tls : This-Layer-Started
Close : administrative Close	tlf : This-Layer-Finished
TO^+ : Timeout with counter > 0	irc : Initialize-Restart-Count
TO^- : Timeout with counter expired	zrc = Zero-Restart-Count
RCR^+ : Receive-Configure-Request (Good)	scr : Send-Configure-Request
RCR^- : Receive-Configure-Request (Bad)	
RCA : Receive-Configure-Ack	sca = Send-Configure-Ack
RCN : Receive-Configure-Nak/Rej	scn = Send-Configure-Nak/Rej
RTR : Receive-Terminate-Request	str = Send-Terminate-Request
RTA : Receive-Terminate-Ack	sta = Send-Terminate-Ack

	0	1	2	3	4	5	6	7	8	9
	Initial	Starting	Closed	Stopped	Closing	Stopping	Req-Sent	Ack-Rcvd	Ack-Sent	Opened
Up	2	irc,scr/6								
Down			0	tls/1	0	1	1	1	1	tld/1
Open	tls/1	1	irc,scr/6	3	5	5	6	7	8	9
Close	0	tlf/0	2	2	4	4	irc,str/4	irc,str/4	irc,str/4	tld,irc,str/4
TO^+					str/4	str/5	scr/6	scr/6	scr/8	
TO^-					tlf/2	tlf/3	tlf/3	tlf/3	tlf/3	
RCR^+			sta/2	irc,scr,sca/8	4	5	sca/8	sca,tlu/9	sca/8	tld,scr,sca/8
RCR^-			sta/2	irc,scr,scn/6	4	5	scn/6	scn/7	scn/6	tld,scr,scn/6
RCA			sta/2	sta/3	4	5	irc/7	scr/6	irc,tlu/9	tld,scr/6
RCN			sta/2	sta/3	4	5	irc,scr/6	scr/6	irc,scr/8	tld,scr/6
RTR			sta/2	sta/3	sta/4	sta/5	sta/6	sta/6	sta/6	tld,zrc,sta/5
RTA			2	3	tlf/2	tlf/3	6	6	8	tld,scr/6

Fig. 3. The PPP Option Negotiation Automaton

the BDD-packages from VIS [7]. The transition predicate is maintained in a conjunctive form. The details are omitted here due to lack of space.

4 Verification of Network Protocols

4.1 Point-to-Point Protocol

Point-to-Point Protocol (PPP) is designed to transmit multi-protocol datagrams for point-to-point communications [22]. To establish the connection, each end sends Link-layer Control Protocol (LCP) packets to configure and test the data link. The authentication may be followed after the link is established. Then PPP sends Network Control Protocol packets to choose and configure network-layer protocols. The link will be disconnected if explicit LCP or NCP packets close it, or certain external events occur (for instance, modem is turned off). In this case study, we focus on checking an implementation of the option negotiation automaton (section 4 in [22]) for link establishment.

Protocol RFC Specification. Figure 3 reproduces the transition table of the automaton as shown in section 4.1 of the specification. As one can see from the table, events and actions are denoted by symbols. For each entry in the table,

```
static void
fsm_rtermack(f)
    fsm *f;
{
    switch (f->state) {
    /* other cases here */
    case OPENED:
        if (f->callbacks->down)
            (*f->callbacks->down)(f);   /* Inform upper layers */
        fsm_sconfreq(f, 0);
        break;
    }
}
```

Fig. 4. Code-style in fsm.c

it shows the actions and the new state of the automaton. If there are multiple actions to be performed in a state, they are executed in an arbitrary order.

When initiating a PPP connection, the host first sends a configuration request packet (scr) to its peer and waits for the acknowledgment (RCA or RCN). The peer responds by checking the options sent in the request. If the options are acceptable, the peer sends a positive acknowledgment (sca). Otherwise, a negative acknowledgment (scn) is sent to the host. In any case, the peer also sends its configuration request packet to the host. They try to negotiate options acceptable to both of them. After they agree on the options, both move to the Opened state and start authentication phrase (or data transmission, if authentication is not required). The communication can be terminated by Close event explicitly or Down event (perhaps due to hardware failure). A termination request (str) is sent if the link is closed explicitly. A restart counter is used to monitor the responses to request actions (scr and str). If the host has not received the acknowledgment from the peer when the timer expires. It sends another request if the counter is greater than zero. Otherwise, it stops the connection locally.

Implementation. The implementation `ppp` version 2.4.0[1] is an open-source package included in various Linux distributions and widely used by Linux users. The package contains several tools for monitoring and maintaining PPP connections as well. The daemon `pppd` implements the protocol and is of our concern here. The file `main.c` uses the subroutines defined in `fsm.c` to maintain the finite state machine. Events and actions have their corresponding subroutines in `fsm.c`. In this work, we assume events and actions are handled correctly. Therefore we leave them as symbols as in the specification. Figure 4 shows how the program behaves on event RTA (receive terminate acknowledgment). For each state that can handle the RTA event, a `case` statement is put in the subroutine. For instance, if RTA is received when the state is Opened, it will inform the

[1] Available at `ftp://ftp.linuxcare.com.au/pub/ppp/ppp-2.4.0.tar.gz`.

upper layers, send a configuration request (`fsm_sconfreq`) and returns. There are 2,589 lines in files `main.c` and `fsm.c`.

Modeling. Once we have defined the constants for events and actions. It is easy to construct a process for the automaton. The following guarded command (written in the language of MOCHA [3]) models the behavior when the state is Opened and the event RTA occurs (figure 3).

```
[] state = Opened & in_p = in_v & evt = RTA & out_p ~= out_v ->
     act' := scr; out_p' := out_v; counter' := dec counter by 1;
     state' := Req_Sent; in_v' := ~in_p
```

The variable `state` denotes the current state, `evt` the event, and `act` the action. The variable `counter` represents the restart counter. It is decremented by one if the action scr is performed. The variables `in_p` and `in_v` model the input channel: the channel is empty if and only if they are equal. Similarly, `out_p` and `out_v` are for the output channel.

For the corresponding implementation (figure 4), more variables are needed to help us for modeling and recovering traces faithfully. We use the variable `addr` to record which subroutine is modeled by the current transition. The boolean variable `timer` is used to model the timeout event: if `timer` is true and the program is in the main loop, it may go to timeout handler. Other variables share the same meaning as those in the specification model.

```
[] addr = rtermack & state = Opened & out_p ~= out_v ->
     act' := scr; out_p' := out_v; timer' := true; counter' := 2;
     in_v' := ~in_p; addr' := input
```

Another process `Link` is used to model the network channel. It accepts an action from one automaton, translates it to an event, and forwards the event to the other automaton. We manually translate the C program to reactive modules. Since the program is well-organized (as seen in figure 4), it may be possible to translate it automatically. The resulting description in MOCHA contains 442 lines of code (182 lines for `pppd` and 260 lines for the specification).

Verification. Having built the models of the specification and implementation, we wish to apply the refinement check. However, certain aspects of the specification are not explicitly present in the implementation. For instance, the automaton is able to send a couple of packets in any order if it is in the state Stopped on event RCR^+ or RCR^-. Two variables are introduced to record which packets have been sent. These variables do not appear in the C program but only in the specification model. As discussed earlier, we need a witness to define these specification variables in terms of the implementation variables. We use the heuristic suggested in [4] to use the eager witness E, and check if $\mathit{pppd} \,\|\, E \preceq \mathit{ppp}$ where *pppd* and *ppp* are the formal models of implementation and specification respectively. However, this refinement relation does not hold. It fails because *pppd* is built with the assumption that it communicates with another PPP automaton. Consequently, we try to establish $\mathit{pppd0} \| \mathit{link} \| \mathit{pppd1} \preceq \mathit{ppp0}$, where *pppd0*, *ppp1* are instances of the implementation model *pppd*, and *link* is the model of the network channel. Using assume-guarantee reasoning, in conjunction with the witness module, this verification goal can be simplified to

$$pppd0 \,||\, link \,||\, ppp1 \,||\, E \;\preceq\; ppp0.$$

This amounts to establishing that the implementation *pppd* refines the specification *ppp* assuming the communication partner satisfies the specification and using E as a witness for the private variables of the specification.

Analysis Result. To check the refinement obligation, we use a prototype built on top of the model checker MOCHA [3]. It produces a trace which describes an erroneous behavior of the implementation. The bug can be seen in the code segment shown in figure 4. On receiving RTA at state Opened, the automaton should bring down the link (tld), send a configuration request (scr) and go to state Req-Sent. However, the implementation does not update the state after it brings down the link and sends the request. In almost all circumstances, the bug is not significant. It can only be observed if the user tries to open the link instantaneously after the disconnection. Our translation lets us trace the bug in the C program easily. After we fix the bug, the refinement relation can be established.

In terms of computational requirements of the refinement check, in comparison to the IWLS image package available in VIS [7], our algorithm requires less memory: while the maximum MDD size with IWLS package is 265,389 nodes, our optimized algorithm the corresponding size is 188,544 nodes, a saving of about 30%. It takes IWLS package 5294.95s to finish while ours for 2318.87s, a saving of 56%.

4.2 Dynamic Host Configuration Protocol

The Dynamic Host Configuration Protocol (DHCP) provides a standard mechanism to obtain configuration parameters. It is widely used in mobile environment, especially for network address allocation. The protocol is designed based on the client-server model. Hosts which provide network parameters are called servers. They are configured by network administrators with consistent information. Clients, on the other hand, communicate with servers and obtain proper parameters to be a host in the network. In a typical scenario, a laptop obtains its network address after it is plugged in any network recognizing DHCP. The user can then access to the network without filling network parameters manually.

The DHCP specification [11] only describes the state machine informally. The state-transition diagram found in section 4.4 [11] gives a global view of the protocol. The details are written in English and scattered around the document. The `dhcp` package version 2.0 patch level 5[2] is the standard implementation distributed by Internet Software Consortium. We are interested in knowing whether the client (`dhclient.c`) is implemented correctly. The implementation does not appear to follow the specification strictly. For instance, it lacks two of the states

[2] Available at `http://www.isc.org/products/DHCP/dhcp-v2.html`.

shown in the state diagram. As a result, it is much more challenging to write down formal models for the specification and implementation in this case than for PPP. We adopt the same style and build four processes: the client specification *client*, the client implementation *dhclient*, the server *server* and the communication channel *link*. Since the implementation performs transitions in several stages, an eager module is introduced to resolve the timing difference. To make the model more realistic, we make the channel *link* lossy. We do not find any inconsistency during the check $dhclient \| link \| server \| E \preceq client$.

In terms of computational requirements of the refinement check, while the maximum MDD size with IWLS package is 13,692 nodes, our optimized algorithm the corresponding size is 29,192 nodes. However, IWLS package takes 350.84s in comparison to 82.70s in our algorithm. It takes 76% less in time in the presence of 53% more in space. We speculate the dynamic ordering algorithm causes this abnormality; further investigation is surely needed.

5 Conclusions

The main contribution of this paper is establishing applicability of refinement checking methodology to verification of implementations of network protocols with respect to RFC documentations. The relevance of the various steps in the methodology is supported by two case studies involving popular protocols, with an inconsistency discovered in one case. We have also proposed a symbolic search algorithm for compressing internal transitions in a hierarchical manner, and established the resulting savings in memory requirements.

In both case studies, the model extraction was done manually. This is unavoidable for extracting specification models since RFC documents typically describe the protocols in a tabular, but informal, format. As far as automating the generation of implementation models from C-code, the emerging technology for model extraction [8, 16, 9, 21] can be useful.

Acknowledgments. We thank Michael Greenwald for many details about PPP and DHCP protocols. This work is partially supported by NSF CAREER award CCR97-34115, by SRC contract 99-TJ-688, by Bell Laboratories, Lucent Technologies, and by 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. R. Alur, L. de Alfaro, R. Grosu, T. Henzinger, M. Kang, R. Majumdar, F. Mang, C. Kirsch, and B. Wang. MOCHA: A model checking tool that exploits design structure. In *Proceedings of 23rd Intl. Conference on Software Engineering*, 2001.
4. R. Alur, R. Grosu, and B.-Y. Wang. Automated refinement checking for asynchronous processes. In *Proc. Third Intl. Workshop on Formal Methods in Computer-Aided Design*. Springer, 2000.

5. R. Alur and T. Henzinger. Reactive modules. *Formal Methods in System Design*, 15(1):7–48, 1999.
6. R. Alur and B.-Y. Wang. "Next" heuristic for on-the-fly model checking. In *CONCUR'99: Concurrency Theory, Tenth Intl. Conference*, LNCS 1664, pages 98–113. Springer, 1999.
7. R. Brayton, G. Hachtel, A. Sangiovanni-Vincentelli, F. Somenzi, A. Aziz, S. Cheng, S. Edwards, S. Khatri, Y. Kukimoto, A. Pardo, S. Qadeer, R. Ranjan, S. Sarwary, T. Shiple, G. Swamy, and T. Villa. VIS: A system for verification and synthesis. In *Proc. Eighth Intl. Conference on Computer Aided Verification*, LNCS 1102, pages 428–432. Springer-Verlag, 1996.
8. J. Corbett, M. Dwyer, J. Hatcliff, S. Laubach, C. Pasareanu, Robby, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *Proceedings of 22nd Intl. Conference on Software Engineering*, pages 439–448. 2000.
9. S. Das, D. Dill, and S. Park. Experience with predicate abstraction. In *Computer Aided Verification, 11th Intl. Conference*, LNCS 1633, pages 160–171. Springer, 1999.
10. J. Fernandez, H. Garavel, A. Kerbrat, R. Mateescu, L. Mounier, and M. Sighireanu. CADP: A protocol validation and verification toolbox. In *Proc. Eighth Intl. Conference on Computer-Aided Verification*, LNCS 1102. Springer-Verlag, 1996.
11. R. Droms. *Dynamic Host Configuration Protocol*, March 1997. RFC 2131.
12. O. Grümberg and D. Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems*, 16(3):843–871, 1994.
13. T. Henzinger, X. Liu, S. Qadeer, and S. Rajamani. Formal specification and verification of a dataflow processor array. In *Proc. Intl. Conference on Computer-aided Design*, pages 494–499, 1999.
14. T. Henzinger, S. Qadeer, and S. Rajamani. You assume, we guarantee: Methodology and case studies. In *CAV 98: Computer-aided Verification*, LNCS 1427, pages 521–525, 1998.
15. G. Holzmann. The model checker SPIN. *IEEE Trans. on Software Engineering*, 23(5):279–295, 1997.
16. G. Holzmann and M. H. Smith. Software model checking - extracting verification models from source code. In *Formal Methods for Protocol Engineering and Distributed Systems*, pages 481–497, Kluwer Academic Publ., 1999.
17. R. Kurshan. *Computer-aided Verification of Coordinating Processes: the automata-theoretic approach*. Princeton University Press, 1994.
18. N. Lynch and M. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proc. Seventh ACM Symposium on Principles of Distributed Computing*, pages 137–151, 1987.
19. K. McMillan. A compositional rule for hardware design refinement. In *CAV 97: Computer-Aided Verification*, LNCS 1254, pages 24–35, 1997.
20. K. McMillan. Verification of an implementation of tomasulo's algorithm by compositional model checking. In *CAV 98: Computer-Aided Verification*, LNCS 1427, pages 110–121, 1998.
21. K. Namjoshi and R. Kurshan. Syntactic program transformations for automatic abstraction. In *Computer Aided Verification, 12th Intl. Conference*, LNCS 1855, pages 435–449. Springer, 2000.
22. W. Simpson. *The Point-to-Point Protocol*. Computer Systems Consulting Services, July 1994. STD 51, RFC 1661.
23. E. Stark. A proof technique for rely-guarantee properties. In *Found. of Software Technology and Theoretical Computer Science*, LNCS 206, pages 369–391, 1985.