Conformance Checking for Models of Asynchronous Message Passing Software

Sriram K. Rajamani and Jakob Rehof

Microsoft Research
{sriram,rehof}@microsoft.com

Abstract. We propose a notion of *conformance* between a specification S and an implementation model I extracted from a message-passing program. In our framework, S and I are CCS processes, which soundly abstract the externally visible communication behavior of a message-passing program. We use the extracted models to check that programs do not get *stuck*, waiting to receive or trying to send messages in vain. We show that our definition of stuckness and conformance capture important correctness conditions of message-passing software. Our definition of conformance was motivated by the need for *modular* reasoning over models, leading to the requirement that conformance preserve substitutability with respect to stuck-freeness: If I conforms to S, and P is any environment such that $P \mid S$ is stuck-free, then it follows that $P \mid I$ is stuck-free. We present a simple algorithm for checking if I conforms to S, when I and S obey certain restrictions.

1 Introduction

Checking behavioral properties of concurrent software using model checking is an active area of current research [7,9,19]. We are interested in extracting message-passing skeletons from concurrent message-passing software, and using model checking to check properties on the skeletons. For this approach to be scalable, we require users to annotate each module with a *behavioral signature*. In our framework [12,3], if F is a module which can communicate with its environment by sending and receiving messages on channels $x_1, \ldots, x_n$, then a signature of F is a CCS process $S(x_1, \ldots, x_n)$ describing the messaging behavior of F on those channels. Each module is then analyzed in isolation to check if its implementation *conforms* with its signature. This requires that a model of the implementation of F is extracted and compared to the signature. Like signatures, implementation models are CCS processes. If I is an implementation model and S is its signature (specification), we use the notation $I \leq S$ to denote that I conforms to S. For our modular reasoning to be sound with respect to a property φ, the conformance relation $\leq$ needs to obey the following *substitutability* property: If $I \leq S$ and P is any environment such that $P \mid S$ satisfies φ then $P \mid I$ satisfies φ ($P \mid S$ denotes the parallel composition of P and S). Substitutability enables a module's behavioral signature to be used instead of a module in invocation contexts, and hence enables model checking to scale.

D. Brinksma and K. G. Larsen (Eds.): CAV 2002, LNCS 2404, pp. 166–179, 2002.

We are interested in checking if message-passing software is *stuck-free*. i.e, that a message sent by a sender will not get stuck without some receiver ever receiving it, and a receiver waiting for a message will not get stuck without some sender ever sending it. Stuck-freeness is a safety property, and in general any safety property [10,17] can be encoded as stuck-freeness. Thus, we desire our conformance relation $\leq$ to satisfy substitutability with respect to stuck-freeness: If $I \leq S$ and P is any environment such that $P \mid S$ is stuck-free, then $P \mid I$ is stuck-free.

A widely used conformance relation is *simulation* [13]. Module I is simulated by module S, written $I \leq^s S$ if every externally visible action α performed by I can also be performed by S such that, if I' and S' are the states of I and S respectively after performing action α, then $I' \leq^s S'$. Simulation does not satisfy substitutability with respect to stuck-freeness due to two reasons: First, while simulation prevents the implementation from doing actions not allowed by the specification, it does not require the implementation to do any particular action. Second, simulation does not distinguish between external nondeterminism that arises due to the ability of the environment to give different inputs, and internal nondeterminism that arises from underspecification or abstraction.

This paper makes the following contributions:

- We present a novel definition of conformance that satisfies substitutability with respect to stuck-freeness. The definition distinguishes between external and internal nondeterminism, requires implementations to do certain actions, and allows implementations that are nonterminating (Section 4).
- We present an algorithm that dualizes specifications using a so-called "mirror construction" to transform a conformance checking problem to a standard invariant checking problem on a product model. Based on this algorithm, a restricted form of conformance checking can be done using an off-the-shelf model checker for invariant checking as a back-end (Section 5).
- We present extensions to the definition of conformance and the checking algorithm to handle cases where the programmer expects certain operations to possibly get stuck, using "weak" send and receive operations (Section 6).

We have implemented a conformance checker as part of a subtype relation checker in a behavioral type system for asynchronous message-passing software. In this paper, we focus entirely on the notion of conformance between CCS models and our algorithm for checking it, assuming that CCS models are extracted from programs. We refer to [12,3] for the problem of model extraction.

2 Background and Related Work

Two traditional notions of conformance between models are language containment [18] and simulation [13]. The former adopts a linear-time view and latter a branching-time view. Our conformance relation is used to define model abstraction in the framework of a behavioral type system [3] where CCS processes [13,14] are used as specifications and models of π-calculus programs [14]. Sound model

abstractions in that system are limited to simulation relations (see [3] for details), and consequently we consider simulation (denoted by $\leq^s$) as a starting point for our investigation.

In this paper, we focus on *stuck-freeness* of communication. Informally, stuck-freeness means that a message sent by a sender will not get stuck without some receiver ever receiving it, and a receiver waiting for a message will not get stuck without some sender ever sending it. This is often an important property in asynchronous software systems. For example, an asynchronous exception may be implemented as a message send operation, and in this case, stuck-freeness of the send operation means that the exception gets caught. Also, asynchronous functions typically get invoked by sending messages to them, and they return results be sending messages back to the caller. In this case, stuck-freeness of the receive operation on the side of the caller means that the caller eventually gets a result.

As mentioned earlier, substitutability is necessary for modular reasoning. Simulation does not preserve substitutability for being stuck-free. For example, let $I_1 = \mathbf{0}$ and $S_1 = x!\#y!$, where $\mathbf{0}$ is the inactive null process, $x!$ denotes a send operation on channel x, and $y!$ denotes a send operation on channel y. The connective $\#$ denotes internal choice. Then $I_1 \leq^s S_1$ holds, but if we consider the environment $P = \texttt{select}\ x? + y?$ (where $\texttt{select}\ x? + y?$ denotes an external choice of a receive operation on channel x and a receive operation on channel y), then $P \mid S_1$ is stuck-free, whereas $P \mid I_1$ is stuck, waiting to receive on either channel x or channel y. The difficulty here is that the implementation does not implement *any* of the internal choices that the specification allows.

As another example, if $I_2 = x?$ (receive on x) and $S_2 = \texttt{select}\ x? + y?$ (external choice of either receive on x or receive on y), we have that $I_2 \leq^s S_2$. Further, for an environment $P = y!$, we have that $P \mid S_2$ is stuck-free, whereas $P \mid I_2$ gets stuck. The difficulty here is that the implementation does not implement *all* of the external choices that the specification prescribes.

The distinction between external and internal choice, and the implications for conformance has been noticed before [1,5]. The notion of alternating simulation proposed in [5] requires the implementation to provide a *subset* of internal choices allowed by the specification and a *superset* of external choices prescribed by the specification. We denote alternating simulation by the symbol $\leq^a$. However, alternating simulation still does not preserve substitutability for being stuck-free. To illustrate this, consider an example with $I_3 = \texttt{select}\ (x?\ \texttt{->}\ y?) + y?$, and $S_3 = x?\texttt{->}y?$. By the definition of $\leq^a$, we have that $I_3 \leq^a S_3$. If $P = (x! \mid y!)$, we have that $P \mid S_3$ is stuck-free but $P \mid I_3$ can get stuck if the interaction on channel y happens first. The difficulty here is that the implementation allows a superset of the external choices prescribed by the specification.

Our definition of conformance was motivated by the above examples. Unlike simulation, we distinguish between internal and external choice [8]. If the specification is an internal choice, then we require the implementation to implement at least one of the choices. If the specification is an external choice, then we require the implementation to implement exactly all of the choices. Further, we allow

arbitrary number of silent internal actions (to allow loops to be abstracted) in the implementation. The formal definition of conformance, and a precise description of how it differs from simulation and alternating simulation can be found in Section 4.

Our notion of conformance and the algorithm for conformance checking are inspired by Dill's work on asynchronous circuit verification [6]. In asynchronous circuits, there is no buffering and an output from one component must be instantaneously received by another component. In asynchronous software, messages are typically queued. Both the conformance relation and the checking algorithm change in subtle ways to reflect this queueing. For example, the process $x! \mid y! \mid x?\texttt{->}y?$ is an erroneous process in Dill's framework since a receiver for the send $y!$ is not immediately available. However, our framework allows $y!$ to block until the interaction on x succeeds and then proceed to have an interaction on y.

An alternative approach to obtain modularity is to reason with context-sensitive abstractions as in [11]. In the context of CSP [8], notions of failures and divergence have been used in model checkers such as FDR [16]. Traces and failures can be used to express various notions of refinement between interfaces as in [2]. Tools have been built to check simulations and bisimulations between CCS processes [4].

3 Models

Our model language is a variant of CCS as presented in [14], where we distinguish between internal and external choice [8]. Our *processes* (ranged over by I, M,M', P, Q, R, S etc.) are defined as follows.

$$\begin{array}{lll} M ::= \mathbf{0} \mid X \mid \texttt{select}\ I_1 + \ldots + I_n \mid O_1 \# \ldots \# O_n & & \text{(Processes)} \\ \quad \mid \ (M_0 \mid M_1) \mid (\nu x)M \mid \mu X.M & & \\ \\ I \ ::= x?\texttt{->}M & & \text{(Guarded expressions)} \\ O \ ::= x!.M & & \end{array}$$

Here, $x!.M$ is a process that sends on channel x and continues as M. Process $x?\texttt{->}M$ receives on channel x and continues as M. We use the symbol $\#$ for internal choice and $+$ for external choice. The process $\texttt{select}\, x?\texttt{->}M + y?\texttt{->}M'$ either receives on x and continues as M, or it receives on y and continues as M'. The process $x!.M \# y!.M'$ continues as either $x!.M$ or as $y!.M'$. We write $x!$ as shorthand for $x!.\mathbf{0}$ and $x?$ as shorthand for $x?\texttt{->}\mathbf{0}$. We also sometimes write $x?\texttt{->}M + y?\texttt{->}M'$ as shorthand for $\texttt{select}\, x?\texttt{->}M + y?\texttt{->}M'$ (*i.e.*, we sometimes leave out the keyword `select`). Label X is used to encode looping in $\mu X.M$. We write $*M$ as an abbreviation for $\mu X.(M \mid X)$.

Given a set of channels $\overline{x}$, the set of *actions* over $\overline{x}$ is defined to be $\{x, x?, x! \mid x \in \overline{x}\} \cup \tau, \epsilon$. The action x is called a *reaction* on channel x. The actions $x?$ and $x!$ are called *commitments* on channel x. Intuitively, two commitments $x?$

Structural Congruence

Structural congruence $\equiv$ is the least congruence relation (equivalence relation closed under term contexts) closed under the following rules, together with renaming and reordering of bound variables and reordering of terms in a summation (either internal or external choice). The set of free names of P is denoted $\mathsf{fn}(P)$.

$$P \mid \mathbf{0} \equiv P \qquad P \mid Q \equiv Q \mid P \qquad P \mid (Q \mid R) \equiv (P \mid Q) \mid R$$

$$(\nu x)\mathbf{0} \equiv \mathbf{0} \qquad \mu X.P \equiv P[\mu X.P/X] \qquad (\nu x)(\nu y)P \equiv (\nu y)(\nu x)P$$

$$\frac{x \notin \mathsf{fn}(P)}{P \mid (\nu x)Q \equiv (\nu x)(P \mid Q)} \qquad \frac{P \equiv P' \quad Q \equiv Q'}{P \mid Q \equiv P' \mid Q'}$$

Labeled Reduction

$$x!.P \mid \texttt{select}\ (\ldots + x?\ \texttt{->}\ Q + \ldots) \xrightarrow{x} P \mid Q \qquad [\textsc{react}]$$

$$x!.P \xrightarrow{x!} P \qquad [\textsc{o-comm}] \qquad \texttt{select}\ (\ldots + x?\ \texttt{->}\ P + \ldots) \xrightarrow{x?} P \qquad [\textsc{i-comm}]$$

$$(\ldots \#x!.P\# \ldots) \xrightarrow{\epsilon} x!.P \qquad [\textsc{i-choice}]$$

$$\frac{P \xrightarrow{x} P'}{(\nu x)P \xrightarrow{\tau} (\nu x)P'} \quad [\textsc{tau}] \qquad \frac{P \xrightarrow{\ell} P' \quad \ell \notin \{x, x?, x!\}}{(\nu x)P \xrightarrow{\ell} (\nu x)P'} \quad [\textsc{res}]$$

$$\frac{P \equiv P' \quad P' \xrightarrow{\ell} Q' \quad Q' \equiv Q}{P \xrightarrow{\ell} Q} \quad [\textsc{s-cong}] \qquad \frac{P \xrightarrow{\ell} P'}{P \mid Q \xrightarrow{\ell} P' \mid Q} \quad [\textsc{par}]$$

$$\frac{P \xrightarrow{\epsilon} P' \quad P' \xrightarrow{\ell} P''}{P \xrightarrow{\ell} P''} \quad [\epsilon\textsc{-left}] \qquad \frac{P \xrightarrow{\ell} P', P' \xrightarrow{\epsilon} P''}{P \xrightarrow{\ell} P''} \quad [\epsilon\textsc{-right}]$$

Eta rules

$$\frac{P \xrightarrow{x} P'}{(\eta x)P \xrightarrow{x} (\eta x)P'} \quad [\textsc{eta}1] \qquad \frac{P \xrightarrow{\ell} P' \quad \ell \notin \{x, x?, x!\}}{(\eta x)P \xrightarrow{\ell} (\eta x)P'} \quad [\textsc{eta}2]$$

Fig. 1. Structural congruence and labeled reduction on CCS processes

and $x!$ on the channel x between two parallel processes can be combined into a reaction x. The action τ is called *silent reaction*, and ϵ is called *null action*.

Figure 1 defines the labeled reduction relation on processes adapted from the commitment and reaction semantics given in [14]. Note that the reduction relation distinguishes between internal and external choice. Internal choice is handled by the rule I-CHOICE where the process chooses the send operation, and external choice is handled by the rule REACT where the environment chooses the receive operation. As indicated by rule S-CONG in Figure 1, reduction is modulo structural congruence. Note that in addition to the usual rules for the restriction operator ν we have rules ETA1 and ETA2 for the restriction operator η. This operator is the same as ν, only with different observability properties: The expression $(\eta\overline{x})P$ is simply meta-notation for a ν-abstraction whose interactions can be observed. This notation is needed to state our substitutability property.

We extend labeled reduction to sequences of actions as well, with ϵ satisfying left and right cancellation under concatenation (rules ϵ-LEFT and ϵ-RIGHT). We write $M_1 \xrightarrow{*} M_2$ if M_1 can transition to M_2 using some sequence of actions. We write M^* to denote the set $\{M' \mid M \xrightarrow{*} M'\}$.

We say that channel x is *bound* in $(\nu x)M$. If x is not bound in M' we say that x is *free* in M'. We say that M is an *end-state* if there are no α and M' such that $M \xrightarrow{\alpha} M'$. If $M \equiv (\eta\overline{x})(P \mid Q)$ is an end-state, and either (1) $P \equiv x!.M_1$, where $x \in \overline{x}$, or (2) $P \equiv$ `select` $(x_{i_1}?\texttt{->} M_1 + x_{i_2}?\texttt{->} M_2 + \ldots + x_{i_k}?\texttt{->} M_k)$ where $\{x_{i_1}, x_{i_2}, \ldots, x_{i_k}\} \in \overline{x}$, then we say that M *is stuck because of* P. M is *stuck* if there exists some P such that M *is stuck because of* P. M is *stuck-free* if no element of M^* is stuck.

Implementation models are defined to be subset of processes satisfying the following property:

A1. For every free channel x, we have x is either used exclusively for sending or exclusively for receiving, and

Assumption A1 is for technical convenience. It is not a fundamental restriction since a bidirectional communication can be achieved using two unidirectional channels, one in either direction. Finally, *Specification models* are defined to be the subset of implementation models that do not contain name restriction, ν.

4 Conformance

Conformance is a binary relation (written $\leq$) between implementation models and specification models. We define conformance in two steps: (1) first we give a definition for specification models that do not have parallel composition, (2) next we extend it to specification models with parallel composition.

Let I be an implementation model and let S be a specification model such that S does not have parallel composition. Informally, I conforms to S if every commitment of I on a free channel can be performed by S, and if S is an internal choice of send operations, then I can perform at least one of the specified send operations, and if S is an external choice of receive operations then I can

perform exactly all of the specified receive operations, and such conformance holds recursively after performing corresponding actions in I and S. The formal definition follows.

Definition 1. *I conforms to S, written $I \leq S$ if I and S have the same set of free channels and there exists a binary relation $H \subseteq I^* \times S^*$ such that the following conditions hold:*

D1. $\langle I, S\rangle \in H$.

D2. For every $\langle P, Q\rangle \in H$ *and every commitment* α *on a free channel we have that if* $P \xrightarrow{\tau^*.\alpha} P'$ *then there exists* $Q' \in Q^*$ *such that* $Q \xrightarrow{\alpha} Q'$ *and* $\langle P', Q'\rangle \in H$.

D3. For every $\langle P, Q\rangle \in H$, *if* $Q = (x_1!.Q_1) \# (x_2!.Q_2) \# \cdots \#(x_n!.Q_n)$, *then for all* P' *such that* $P \xrightarrow{\tau^*} P'$, *either there exists* P'' *such that* $P' \xrightarrow{\tau} P''$, *or there exists* $1 \leq i \leq n$ *and* $P_i \in I^*$ *such that* $P' \xrightarrow{x_i!} P_i$, *and* $\langle P_i, Q_i\rangle \in H$.

D4. For every $\langle P, Q\rangle \in H$, *if* $Q = (x_1?\texttt{->}Q_1) + (x_2?\texttt{->}Q_2) + \cdots + (x_n?\texttt{->}Q_n)$, *then for all* P' *such that* $P \xrightarrow{\tau^*} P'$, *either there exists* P'' *such that* $P' \xrightarrow{\tau} P''$, *or for all* $1 \leq i \leq n$ *there exists* $P_i \in I^*$ *such that* $P' \xrightarrow{x_i?} P_i$, *and* $\langle P_i, Q_i\rangle \in H$.

We extend conformance to specification models with parallelism by requiring structural similarity in the implementation and specification: we say that $I_1 \mid I_2 \leq S_1 \mid S_2$ if $I_1 \leq S_1$ and $I_2 \leq S_2$.

Properties Conformance is stronger than both simulation ($\leq^s$) and alternating simulation ($\leq^a$). Simulation is obtained by removing conditions D3 and D4 from the definition. Alternating simulation is obtained by removing condition D3, and weakening condition D2 to consider only send commitments on free channels. We recall some examples from Section 2 to illustrate the differences between simulation, alternating simulation and conformance. If $I_1 = \mathbf{0}$ and $S_1 = x!\#y!$, we have that $I_1 \leq^s S_1$, but $I_1 \not\leq S_1$ since condition D3 is violated. If $I_2 = x?$ and $S_2 = x? + y?$, we again have that $I_2 \leq^s S_2$, but $I_2 \not\leq S_2$ since condition D4 is violated. Finally, if $I_3 = \texttt{select}\ (x?\ \texttt{->}\ y?) + y?$, and $S_3 = x?\ \texttt{->}\ y?$, we have that $I_3 \leq^a S_3$, but $I_3 \not\leq S_3$ since condition D2 is violated for commitment $y?$ from I_3.

We also note that simulation and alternating simulation are incomparable. For example, $I_2 \leq^s S_2$, but $I_2 \not\leq^a S_2$, and $I_3 \leq^a S_3$, but $I_3 \not\leq^s S_3$.

We have intentionally stated the definition for conformance to allow the implementation to perform an arbitrary number of internal τ actions. This is in accordance with usual practice in typed programming languages, which allows nonterminating implementations. For example, most type systems will allow the function `foo(x) = while(true) {skip}; return(x+1)` with a nonterminating while loop to be typed with the type signature $int \rightarrow int$.

Conformance is not the largest relation satisfying substitutability for being stuck-free. For example, with $I = x? \mid y?$ and $S = \texttt{select}\ x?\ \texttt{->}\ y?$, we have that any environment P such that $P \mid I$ gets stuck would also make $P \mid S$ stuck.

The example shows that it is sometimes possible to sequentialize an implementation in the specification with respect to stuck-freeness. Alternating simulation would accept $I \leq^a S$, but as shown above, $\leq^a$ does not always respect substitutability for stuck-freeness. The following theorem states that conformance obeys substitutability for being stuck-free. A proof of the theorem can be found in [15].

Theorem 1. *Let I be any implementation model and S be any specification model, with the same set of free channels $\overline{x}$. If $I \leq S$, then for any process P if $(\eta\overline{x})(P \mid S)$ is stuck-free, then $(\eta\overline{x})(P \mid I)$ is stuck-free.*

Example Suppose a service F can be called with channel parameters x , y and e for sending back responses to the caller. Suppose further that F asynchronously calls two other services G and H (using response channels z and w, respectively) to produce its responses on x and y, respectively. Finally, assume that H may either respond normally or raise an exception by sending a message on a channel Err, in which case F will throw an exception by sending on e. An implementation of F could be as shown below, declared with behavioral signature $x!.(y!\#e!)$. We wish to decide whether F conforms to its signature, given only the signatures (and no implementations) for G and H.

```
G(z) : z!
H(w) : w! # Err!

F(x,y) : x!.(y! # e!)
{
  try{
    async(z,w){G(z), H(w)}{
       select z? -> { send x; select w? -> send y;}}
  } catch Err {throw e}
}
```

A model for the body of F is constructed by substituting the signatures for G and H at the call sites, yielding the CCS model M_F shown below. Notice that the asynchronous calls to G and H are run concurrently, and the exception handler is modeled by waiting to receive on Err in the select statements. Because the channels z and w are used internally by F to receive responses from G and H, these channels are restricted using the ν operator.

$$M_F = (\nu z)(\nu w)(z! \mid (w!\#Err!) \mid \\ (\texttt{select}\ (z?\ \texttt{->}\ x!.(\texttt{select}\ (w?\ \texttt{->}\ y!) + (Err?\ \texttt{->}\ e!)) + (Err?\ \texttt{->}\ e!)))$$

Checking conformance of F to its signature, we decide whether $M_F \leq x!.(y!\#e!)$. We find that the conformance test fails: if H throws an exception before F has sent its response on x, then no response will be sent on x at all. An environment expecting, according to the signature of F, a response on x followed by

a message on either y or e, will be stuck. The problem can be fixed either by changing the signature of F or by changing the implementation of F. A signature that would work is $(x!.(y!\#e!))\#e!$. This signature does not promise much to the environment, and it suggests that one would rather want to change the implementation of F to conform to its original signature. This could be done by sequentializing the calls to G and H, by first waiting for G to respond and only then calling H afterwards. Or, perhaps better, one could throw the exception e only after sending on x. Notice that, by restricting z and w used for internal communication with G and H in F, conformance allows us to avoid exposing that communication in the signature of F. This is essential for a scalable module system.

5 Mirrors

Given an implementation model I and a specification model S, we would like to check if $I \leq S$ using a state space exploration on I. If S and I obey certain restrictions, then there is a simple and elegant way to do this by constructing a so called *mirror* of S and a so called *serialization* of I. Using this construction we can perform conformance checking using any off-the-shelf model checker.

In order to describe the necessary restrictions for our mirror construction, we impose two restrictions on processes:

A2. Specifications and implementations do not have mixed selects. Formally, every external choice of the form $(x_1?\texttt{->}M_1 + x_2?\texttt{->}M_2 + \cdots + x_k?\texttt{->}M_k)$, either all the x_i are free channels or all the x_i are bound channels.

A3. Specifications do not contain hidden internal nondeterminism. Formally, the condition means that for all T_1 such that $T \xrightarrow{*} T_1$, we have that for every commitment α, whenever $T_1 \xrightarrow{\alpha} T_2$ and $T_1 \xrightarrow{\alpha} T_3$, it is the case that $T_2 \equiv T_3$.

Finally, we strengthen the conformance relation $\leq$ to a stronger relation $\preceq$ by strengthening condition D4. in Definition 1 as follows:

D4'. For every $\langle P, Q\rangle \in H$, if $Q = (x_1?\texttt{->}Q_1)+(x_2?\texttt{->}Q_2)+\cdots+(x_n?\texttt{->}Q_n)$, then for all P' such that $P \xrightarrow{\tau^*} P'$, either there exists P'' such that $P' \xrightarrow{\tau} P''$, or $P' = (x_1?\texttt{->}P_1) + (x_2?\texttt{->}P_2) + \cdots + (x_n?\texttt{->}P_n)$ such that $\langle P_i, Q_i\rangle \in H$ for $1 \leq i \leq n$.

Since D4' is stronger than D4, the strengthened relation $\preceq$ satisifies Thorem 1. We give an algorithm for checking $\preceq$ below.

For a specification S without parallel composition, we can construct a *mirror* process $\mathcal{M}(S)$ that represents all possible environments that S can potentially be composed with. Let $\overline{x} = \{x_1, x_2, \ldots, x_n\}$ be the set of free channels in the specification. The definition of $\mathcal{M}(.)$ is given in Figure 2, by structural induction on the specification. The definition uses an error process *Err* which is used to denote one kind of failure in the conformance check.

$$
\begin{array}{ll}
\mathcal{M}(\mathbf{0}) & = \mathbf{0} \\
\mathcal{M}\left((x_{i_1}?\texttt{->}P_1) + \ldots + (x_{i_k}?\texttt{->}P_k)\right) & = (x_{i_1}!.\mathcal{M}(P_1)) \# \ldots \# (x_{i_k}!.\mathcal{M}(P_k)) \\
 & \quad \#(\underline{x_{j_1}}!.Err \# \underline{x_{j_2}}!.Err \# \underline{x_{j_m}}!.Err) \\
 & \quad \text{where } \{x_{j_1}, \ldots, x_{j_m}\} = \overline{x} \setminus \{x_{i_1}, \ldots, x_{i_k}\} \\
\mathcal{M}\left((x_{i_1}!.P_1) \# \ldots \# (x_{i_m}!.P_m)\right) & = (x_{i_1}?\texttt{->}\mathcal{M}(P_1)) + \ldots + (x_{i_m}?\texttt{->}\mathcal{M}(P_m)) \\
\mathcal{M}(X) & = X \\
\mathcal{M}(\mu X.M) & = \mu X.\mathcal{M}(M)
\end{array}
$$

Fig. 2. Mirror construction

For an implementation model I, with free channels $\overline{x}$, we construct a *serialized* process $\mathcal{R}(S)$ that serializes all communications on free channels using a lock implemented by two global channels *acq* and *rel*. The definition of the serialization function $\mathcal{R}(.)$ is given in Figure 3, by structural induction on the implementation.

Note that our assumption A2. that external choices are either over free channels, or over bound channels, but not a mixture of both free and bound channels is used in the definition of $\mathcal{R}(.)$. Let $L = \mu X.acq!.rel?\texttt{->}X$ be a process that implements a lock. Given an implementation I and specification S, in order to check if $I \preceq S$, we check if $\mathcal{M}(I) \mid \mathcal{R}(S) \mid L$ neither gets stuck in certain ways (mentioned below) nor reaches the error process Err.

Before making this statement precise, we need a weaker notion of stuckness. We say that M is a *τ-end-state* if for all $M \xrightarrow{\omega} M'$, we have that $\omega \in \tau^*$. If $M \equiv (\eta\overline{x})(P \mid Q)$ is a τ-end-state, and either (1) $P \equiv x!.M_1$, where $x \in \overline{x}$, or (2) $P \equiv \texttt{select}(x_{i_1}?\texttt{->}M_1 + x_{i_2}?\texttt{->}M_2 + \ldots + x_{i_k}?\texttt{->}M_k)$ where $\{x_{i_1}, x_{i_2}, \ldots, x_{i_k}\} \in \overline{x}$, then we say that M is *τ-stuck because of P*. M is *τ-stuck* if there exists some P such that M *is τ-stuck because of P*.

The following theorem contains our algorithm for checking conformance $I \preceq S$ via the process $\mathcal{M}(I) \mid \mathcal{R}(S) \mid L$. A proof of the theorem can be found in [15].

Theorem 2. *Let I be any implementation model and S be any specification model, with the same set of free channels $\overline{x}$. Let $L() = \mu X.acq!.rel?\texttt{->}X$ be a process that implements a lock using two channels acq and rel that are not present in I or S. We have that $I \preceq S$ iff the following three conditions hold for all processes $\hat{M} \equiv (\eta(\overline{x}, acq, rel))(\mathcal{M}(\hat{S}) \mid \mathcal{R}(\hat{I}) \mid \hat{L})$, such that $(\eta(\overline{x}, acq, rel))(\mathcal{M}(S) \mid \mathcal{R}(I) \mid L) \xrightarrow{*} (\eta(\overline{x}, acq, rel))(\mathcal{M}(\hat{S}) \mid \mathcal{R}(\hat{I}) \mid \hat{L})$:*

C1. $\hat{S} \not\equiv Err$
C2. $\hat{M}$ is not τ-stuck because of $\mathcal{R}(\hat{I})$
C3. $\hat{M}$ is not stuck because of $\mathcal{M}(\hat{S})$

$$\mathcal{R}(\mathbf{0}) = \mathbf{0}$$

$$\mathcal{R}\left((x_{i_1}?\text{->}P_1) + \ldots + (x_{i_k}?\text{->}P_k)\right) = acq?\text{->}\left(\begin{array}{l}(x_{i_1}?\text{->}rel!.\mathcal{R}(P_1)) + \ldots + \\ (x_{i_k}?\text{->}rel!.\mathcal{R}(P_k))\end{array}\right) \quad \text{if } \{x_{j_1}, \ldots, x_{j_m}\} \subseteq \overline{x}$$

$$\mathcal{R}\left((y_{i_1}?\text{->}P_1) + \ldots + (y_{i_k}?\text{->}P_k)\right) = \left((y_{i_1}?\text{->}\mathcal{R}(P_1)) + \ldots + (y_{i_k}?\text{->}\mathcal{R}(P_k))\right) \quad \text{if } \{y_{j_1}, \ldots, y_{j_m}\} \cap \overline{x} = \phi$$

$$\mathcal{R}\left((z_{i_1}!.P_1)\#\ldots\#(z_{i_m}!.P_m)\right) = \left((\mathcal{T}(z_{i_1}).\mathcal{R}(P_1)\#\ldots\#(\mathcal{T}(z_{i_m}).\mathcal{R}(P_m)\right), \quad \text{where } \mathcal{T}(z_{i_k}) = \begin{array}{l} z_{i_k}! \text{ if } z_{i_k} \notin \overline{x}, \text{ and} \\ acq?\text{->}z_{i_k}!.rel! \text{ if } z_{i_k} \notin \overline{x}\end{array}$$

$$\mathcal{R}(X) = X$$

$$\mathcal{R}(\mu X.M) = \mu X.\mathcal{R}(M)$$

$$\mathcal{R}(M_0 \mid M_1) = \mathcal{R}(M_0) \mid \mathcal{R}(M_1)$$

$$\mathcal{R}((\nu x)M) = (\nu x)\mathcal{R}(M)$$

Fig. 3. Serialization

6 Weak Send and Receive

Our definition of stuckness so far assumes that send and receive operations must always succeed. This is not always feasible. A parallel search program spawns off a number of child processes, and after the first process to discover the item reports success, the remaining processes are disregarded by the parent. We do not wish to regard the program as stuck, even though all child processes but one could be sending messages in vain. We distinguish between operations that are intended to succeed and operations that are allowed not to succeed by referring to the latter as *weak send* and *weak receive* operations, respectively, and we introduce special syntax for the weak operations. This allows us to treat them differently in checking conformance. We introduce the extended syntax for send and receive operations:

$$\begin{array}{rl} M & ::= \ldots \mid \underline{\texttt{select}}\, I_1 + \ldots + I_n \\ O & ::= \ldots \mid \underline{x}!.M \end{array}$$

Here, a *weak receive* is of the form $\underline{\texttt{select}}\, I_1 + \ldots + I_n$, and allows all of the receives specified in I_1 through I_n to not succeed. Similarly, a *weak send* $\underline{x}!.M$ is allowed not to succeed. Notice that we allow internal choices of mixed mode. For example, the expression $x!.M_1 \,\#\, \underline{y}!.M_2$ is allowed, and if the left branch is chosen, then it leads to the operation $x!.M_1$ where the send is required to succeed, whereas choosing the right branch leads to $\underline{y}!.M_2$, which may or may not succeed.

We need to extend our notions of conformance ($\leq$) and conformance checking for the weak constructs. Consider first a weak send, $\underline{x}!$. We require that $\underline{x}! \leq x!$ holds, but on the other hand, $x! \not\leq \underline{x}!$. Intuitively, the operation $x!$ requires more of any operating environment than the operation $\underline{x}!$: the former operation gets *stuck* in an environment that does not offer the corresponding receive on x. Hence, substitutability is preserved for this extension of $\leq$. Similarly, we require that $\underline{\texttt{select}}\ I \leq \texttt{select}\ I$ holds, but on the other hand $\texttt{select}\ I \not\leq \underline{\texttt{select}}\ I$.

We extend our conformance check to handle the weak operations by extending the notion of mirrors, as follows:

$$\mathcal{M}\left(\underline{\texttt{select}}\,(I_1 + \ldots + I_n)\right) = \mathcal{M}\left(\texttt{select}\ (I_1 + \ldots + I_n)\right) \#\mathbf{0}$$

$$\mathcal{M}\left((\xi_1!.P_1)\#\ldots\#(\xi_n!.P_n)\right) = \texttt{select}\ (\tilde{\xi}_1?\texttt{->}\mathcal{M}(P_1)) + \ldots + (\tilde{\xi}_n?\texttt{->}\mathcal{M}(P_n))$$

where ξ ranges over channels of the usual form x or of the weak form $\underline{x}$, and where we define $\tilde{\xi} = \tilde{x}$, if $\xi = \underline{x}$, and $\tilde{\xi} = x$, if $\xi = x$. In this defintion, the special receive form $\texttt{select}\ (\tilde{x}?\texttt{->}\mathcal{M}(P))$ (with action $\tilde{x}?$) is used to mirror the weak send. This form is used internally by the conformance checker and defines an extended notion of *stuckness*, capturing the conformance relation with weak send. Thus, the process $\texttt{select}(\tilde{x}?\texttt{->}\mathbf{0}) \mid x!.\mathbf{0}$ is classified as stuck, corresponding to the fact that the conformance relation $x! \leq \underline{x}!$ does *not* hold. On the other hand, the process $\texttt{select}\ (\tilde{x}?\texttt{->}\mathbf{0}) \mid \underline{x}!.\mathbf{0}$ is not classified as stuck (and it reacts normally on x), so the conformance relation $\underline{x}! \leq x!$ holds. We also change the notion of stuckness to allow for weak receives with no corresponding send. For example, we do not classify $\underline{\texttt{select}}\ (x?\texttt{->}\mathbf{0}) \mid \mathbf{0}$ as stuck. The mirror of the weak receive, as defined above, adds the choice of the null process $\mathbf{0}$ to the previous definition of $\mathcal{M}$. Hence, weak receive specification $\underline{\texttt{select}}\ I$ requires implementation to be prepared to handle the situation where no input arrives. By our previous notion of stuckness, we have that $\texttt{select}\ (x?\texttt{->}\mathbf{0}) \mid \mathbf{0}$ is stuck. It follows (by our definitions above) that

$$\texttt{select}\ (x?\texttt{->}\mathbf{0}) \mid \mathcal{M}(\underline{\texttt{select}}\,(x?\texttt{->}\mathbf{0})) = \texttt{select}\ (x?\texttt{->}\mathbf{0}) \mid x!.\mathbf{0}\#\mathbf{0}$$

can evolve to an end-state, so the conformance relation $\texttt{select}\ (x?\texttt{->}\mathbf{0}) \leq \underline{\texttt{select}}\,(x?\texttt{->}\mathbf{0})$ does *not* hold.

7 Conclusion

Scalable model checking of models extracted from message-passing software requires that components be abstracted by simpler specifications. Existing definitions of conformance between a component and its abstraction —simulation and alternating simulation— do not satisfy substitutability for stuck-freeness. We presented a new definition of conformance that has this property. Our definition of conformance is stronger than both simulation and alternating simulation. We gave a simple algorithm to check if an implementation I conforms to a specification S, assuming certain restictions on I and S, by serializing I, dualizing S

and doing a state space exploration on the product. We have implemented the algorithm using the SPIN model checker, as a back-end for a behavioral type checker for message-passing software.

Acknowledgments

We thank Cedric Fournet, Tony Hoare, Gerard Holzmann, Jim Larus and Bill Roscoe for very helpful discussions.

References

1. R. Alur, T. A. Henzinger, O. Kupferman, and M. Y. Vardi. Alternating refinement relations. In *CONCUR 98: Concurrency Theory*, LNCS 1466, pages 163–178. Springer-Verlag, 1998. 168
2. E. Brinksma, B. Jonsson, and F. Orava. Refinining interfaces of communicating systems. In *TAPSOFT 91: Theory and Practice of Software Development*, LNCS 494, pages 297–312. Springer-Verlag, 1991. 169
3. S. Chaki, S. K. Rajamani, and J. Rehof. Types as models: Model checking message-passing programs. In *POPL 02: ACM Principles of Programming Languages*, pages 45–57. ACM, 2002. 166, 167, 168
4. R. J. Cleaveland, J. Parrow, and B. Steffen. The Concurrency Workbench: a semantics-based tool for the verification of finite-state systems. *ACM Transactions on Programming Languages and Systems*, 15(1):36–72, 1993. 169
5. L. de Alfaro and T. A. Henzinger. Interface theories for component-based design. In *EMSOFT 01: Embedded Software*, LNCS, pages 148–165. Springer-Verlag, 2001. 168
6. D. L. Dill. *Trace Theory for Automatic Verification of Speed-Independent Circuits*. MIT Press, 1988. 169
7. M. Dwyer, J. Hatcliff, R. Joehanes, S. Laubach, C. Pasareanu, Robby, W. Visser, and H. Zheng. Tool-supported program abstraction for finite-state verification. In *ICSE 01: International Conference on Software Engineering*, pages 177–187. ACM, 2001. 166
8. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985. 168, 169
9. G. J. Holzmann. Logic verification of ANSI-C code with Spin. In *SPIN 00: SPIN Workshop*, LNCS 1885, pages 131–147. Springer-Verlag, 2000. 166
10. L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, SE-3(2):125–143, 1977. 167
11. Kim G. Larsen and Robin Milner. A compositional protocol verification using relativized bisimulation. *Information and Computation*, 99:80–108, 1992. 169
12. J. R. Larus, S. K. Rajamani, and J. Rehof. Behavioral types for structured asynchronous programming. Technical report, Microsoft Research, 2001. 166, 167
13. R. Milner. *Communication and Concurrency*. Prentice Hall, 1989. 167
14. R. Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, 1999. 167, 169, 171
15. S. K. Rajamani and J. Rehof. Conformance checking for models of asynchronous message passing software. Technical report, Microsoft Research, 2002. 173, 175
16. A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1998. 169

17. F. B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, February 2000. 167
18. M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *LICS 86: Logic in Computer Science*, pages 322–331. IEEE Computer Society Press, 1986. 167
19. W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *ICASE 00: Automated Software Engineering*, pages 3–12, 2000. 166