Stuck-Free Conformance

Cédric Fournet, Tony Hoare, Sriram K. Rajamani, and Jakob Rehof

Microsoft Research
{fournet,thoare,sriram,rehof}@microsoft.com

Abstract. We present a novel refinement relation (stuck-free conformance) for CCS processes, which satisfies the substitutability property: If I conforms to S, and P is any environment such that $P \mid S$ is stuck-free, then $P \mid I$ is stuck-free. Stuck-freedom is related to the CSP notion of deadlock, but it is more discriminative by taking orphan messages in asynchronous systems into account. We prove that conformance is a precongruence on CCS processes, thereby supporting modular refinement. We distinguish conformance from the related preorders, stable failures refinement in CSP and refusal preorder in CCS. We have implemented conformance checking in a new software model checker, ZING, and we report on how we used it to find errors in distributed programs.

1 Introduction

We are interested in checking that message-passing programs are *stuck-free* [12]. Stuck-freedom formalizes the property that a communicating system cannot deadlock waiting for messages that are never sent or send messages that are never received. In this paper we extend [12] by generalizing the theory of conformance and by reporting on its application in model checking distributed programs. In our example application, programmers write *contracts*, which are interfaces that specify the externally visible message-passing behavior of the program. Contracts can be as rich as CCS processes. Stuck-freedom is ensured by checking that an implementation *conforms* to its contract using a model checker.

Checking stuck-freedom by exploring the state space of the entire system quickly leads to state explosion, because the state space grows exponentially in the number of concurrent processes; and it requires that the entire system is available for analysis, which is especially unrealistic for distributed systems. We therefore wish to check stuck-freedom compositionally. If I is an implementation of a component and C is its contract, we use the notation $I \leq C$ to denote that I conforms to C. For our compositional approach to be sound with respect to stuck-freedom, the conformance relation $\leq$ needs to obey the following *substitutability* property: If $I \leq C$ and P is any environment such that $P \mid C$ is stuck-free, then $P \mid I$ is stuck-free as well ($P \mid C$ denotes the parallel composition of P and C). Substitutability means that the contract of a component can be safely used instead of the component in invocation contexts, and hence it helps model checking to scale.

Our notion of conformance is a novel process preorder that preserves *stuckness.* Stuckness can be directly observed in any labeled transition system in which visible actions as well as stability (the inability to perform hidden actions) can be observed. In our applications this is important, because we want to analyze a system by executing a model of it and observing what happens. Stuckness is more discriminative than CSP deadlock or unspecified reception [10], since stuckness encompasses any "left-over" action on a channel name. In addition to deadlock, this includes orphan messages that are never consumed in asynchronous systems and directly models important failure conditions in software such as unhandled exception messages.

This paper makes the following contributions:

– We define a notion of conformance based on standard CCS transition semantics and prove that it is a precongruence (i.e., it is preserved by all CCS contexts) satisfying the substitutability property for stuck-freedom. We distinguish conformance from the most closely related known preorders, CSP stable failures refinement [3, 6, 13] and CCS refusal preorder [11].
– We have implemented a conformance checker in our software model checker, ZING. The implementation technique is quite general and can be used to adapt existing model checkers for software to do conformance checking. ZING processes can be regarded as CCS processes, and hence our conformance theory applies to the ZING conformance checker. We have applied ZING to check contract conformance in a non-trivial distributed application, leading to the discovery of several bugs in the application.

This paper significantly extends the work reported in [12], which proposes a conformance relation satisfying the substitutability property. The conformance relation of [12] is limited in several respects. First, conformance in [12] is defined by reference to the syntactic form of processes[1], which precludes a purely observational implementation of conformance checking by recording actions generated by transitions. Second, the relation in [12] is not a precongruence, because the syntactic constraints are not preserved under arbitrary CCS contexts. As a consequence, natural and useful algebraic laws fail, including the desirable modular principle, that $P_1 \leq Q_1$ and $P_2 \leq Q_2$ implies $P_1 \mid P_2 \leq Q_1 \mid Q_2$. Third, the theory in [12] is complicated by using a non-standard action[2] to treat nondeterminism. This paper provides a substantial generalization of conformance that is purely observational, is based on the standard CCS transition system, and gives a unified treatment of nondeterminism, hiding, and stability in terms of hidden actions (τ). It can be implemented in a model checker by observing visible actions and stability, and it can be compared to other semantic refinement relations. In addition to proving substitutability, we prove that our generalized conformance relation is a precongruence, thereby supporting modular refinement. Finally, we have applied this theory by implementing it in a software model checker and applying it to contract checking for distributed programs.

[1] For example, some cases in the conformance definition of [12] apply only to processes of the form $P\#Q$ or $P + Q$. Other similar syntactic dependencies of this nature is present in [12] as well.

[2] The action is named ϵ in [12].

The remainder of this paper is organized as follows. In Section 2 we discuss refinement relations in the CSP and CCS traditions that are most closely related to stuck-free conformance. In Section 3 we present our theory of conformance. In order to keep the paper within limits, we have left out all proofs in the presentation of our conformance theory. Fully detailed proofs can be found in our technical report [5]. In Section 4 we describe the application of the ZING conformance checker to find errors in distributed programs, and Section 5 concludes.

2 Related Work

Our notion of conformance is modeled on stable failures refinement in CSP [3, 6, 13] and is inspired by the success of the refinement checker FDR [13]. Our process model combines the operational semantics of CCS with a notion of refusals similar to but distinct from the CSP notion. The definition of conformance relies on simulation and applies directly to any labeled transition system in which visible actions and stability can be observed. As is discussed in Section 3.3, substitutability with respect to stuck-freedom is not satisfied, if stable failures refinement is adopted directly into CCS. The reason is that stuck-freedom is a different and more discriminative notion than the CSP concept of deadlock. In order to accomodate this difference, our conformance relation is based on the idea of *ready refusals*, which requires a process to be ready to accept certain actions while (at the same time) refusing others.
Stuck-free conformance is also related to the refusals preorder for CCS as developed in Iain Phillips' theory of refusal testing [11]. Refusal preorder allows the observation of actions that happen after refusals and can therefore express readiness conditions in combination with refusals. However, as discussed in Section 3.4, stuck-free conformance is a strictly larger precongruence than refusal preorder.
The notion of 2/3 bisimulation, presented by Larsen and Skou [7] bears some resemblance to conformance. However, Larsen and Skou do not treat hiding and internal actions (τ-actions), and therefore the problems of stability do not arise there. Our theory of conformance gives a unified treatment of internal actions, non-determinism, and stability, which are essential in our applications.
Alternating simulation [1] has been used to relate interfaces to implementations in [4]. As discussed in [12], alternating simulation does not satisfy substitutability for stuck-freedom.

3 Conformance Theory

In this section we give necessary background on CCS [8, 9], we define our notions of stuck-freedom and conformance, and we prove that conformance is a precongruence (Theorem 1) and that it satisfies the substitutability property (Theorem 2).

3.1 CCS Processes

We assume a denumerable set of names $\mathcal{N} = \{a, b, c, \ldots\}$. The set $\mathcal{L} \stackrel{\Delta}{=} \mathcal{N} \cup \{\bar{a} \mid a \in \mathcal{N}\}$ is called the set of *labels*. The set $\mathcal{A} \stackrel{\Delta}{=} \mathcal{L} \cup \{\tau\}$ is called the set of *actions*.

We let α range over $\mathcal{A}$ and we let λ range over $\mathcal{L}$, and we write $\bar{\bar{a}} = a$. For a subset X of $\mathcal{L}$ we write $\bar{X} = \{\bar{a} \mid a \in X\}$. CCS processes, ranged over by P, are defined by:

$$\begin{aligned} P &::= \mathbf{0} \mid A\langle a_1, \ldots, a_n\rangle \mid G_1 + \ldots + G_n \mid (P|Q) \mid (\nu\ a)P \\ G &::= \alpha.P \end{aligned}$$

Here, A ranges over process names, with defining equations $A \stackrel{\Delta}{=} P$. We say that the name a is bound in $(\nu\ a)P$. The free names of P, denoted $\mathrm{fn}(P)$, are the names in P that are not bound.

Definition 1 (Structural congruence, $\equiv$). *Structural congruence, $\equiv$, is the least congruence relation on terms closed under the following rules, together with change of bound names and variables (alpha-conversion) and reordering of terms in a summation:*

1. $P|\mathbf{0} \equiv P$, $P|Q \equiv Q|P$, $P|(Q|R) \equiv (P|Q)|R$
2. $(\nu\ a)(P|Q) \equiv P|(\nu\ a)Q$, *if* $a \notin fn(P)$
3. $(\nu\ a)\mathbf{0} \equiv \mathbf{0}$, $(\nu\ ab)P \equiv (\nu\ ba)P$, $(\nu a)(\nu b)P \equiv (\nu b)(\nu a)P$

The operational semantics of CCS is given by the labeled transition system shown in Definition 2. This system is exactly the one given by Milner [9], except that we have added rule [CONG], as is standard in recent presentations of CCS. In rule [SUM] below, M ranges over a summation.

Definition 2 (Labeled transition).

$$M + \alpha.P \xrightarrow{\alpha} P \quad [\textsc{sum}] \qquad \frac{P \xrightarrow{\lambda} P' \quad Q \xrightarrow{\bar{\lambda}} Q'}{P|Q \xrightarrow{\tau} P'|Q'} \quad [\textsc{react}]$$

$$\frac{P \xrightarrow{\alpha} P'}{P|Q \xrightarrow{\alpha} P'|Q} \quad [\textsc{par-l}] \qquad \frac{P \equiv P' \quad P' \xrightarrow{\alpha} Q' \quad Q' \equiv Q}{P \xrightarrow{\alpha} Q} \quad [\textsc{cong}]$$

$$\frac{P \xrightarrow{\alpha} P' \quad \alpha \notin \{a, \bar{a}\}}{(\nu\ a)P \xrightarrow{\alpha} (\nu\ a)P'} \quad [\textsc{res}]$$

$$\frac{P_A[\tilde{b}/\tilde{a}] \xrightarrow{\alpha} P' \qquad A(\boldsymbol{a}) \stackrel{\Delta}{=} P_A}{A\langle\tilde{b}\rangle \xrightarrow{\alpha} P'} \quad [\textsc{ident}]$$

We let $P \# Q \stackrel{\Delta}{=} \tau.P + \tau.Q$. The difference between $a.P \# b.Q$ and $a.P + b.Q$ is important. The former represents *internal choice*, where the process chooses to transition to $a.P$ or $b.Q$, whereas the latter represents *external choice* [6], where the environment controls whether the process moves to P or Q by offering $\bar{a}$ or $\bar{b}$. The distinction between internal and external choice is crucial in our notion of conformance.

Asynchronous actions are modeled as actions with no sequential continuation in CCS. Hence, asynchronous actions are "spawned", as in $a \mid P$, where a happens asynchonously.

Notation. We write $\tilde{a}$ and $\tilde{\alpha}$ for (possibly empty) sequences of names and actions. For $\tilde{\alpha} = \alpha_0 \ldots \alpha_{n-1}$ we write $P \xrightarrow{\tilde{\alpha}} P'$, if there exist $P_0, P_1, \ldots, P_n$ such that $P \equiv P_0$, $P' \equiv P_n$ and for $0 \leq i < n$, we have $P_i \xrightarrow{\alpha_i} P_{i+1}$. We use the notation $P \xrightarrow{\tau^* \lambda} P'$, where $(\tau^* \lambda) \in \{\lambda, \tau\lambda, \tau\tau\lambda, \ldots\}$. We write $P \longrightarrow P'$ if there exists $\tilde{\alpha}$ such that $P \xrightarrow{\tilde{\alpha}} P'$; $P \xrightarrow{\tilde{\alpha}}$ means there exists P' such that $P \xrightarrow{\tilde{\alpha}} P'$; $P \longrightarrow$ means that $P \xrightarrow{\tilde{\alpha}}$ for some $\tilde{\alpha}$. P is *stable* if P can make no hidden actions, i.e., $P \not\xrightarrow{\tau}$, and P is an *end-state* if P can make no action at all, i.e., $P \not\longrightarrow$. Finally, we use the shorthand notation $\lambda \bar{\in} \tilde{a}$ to mean that either λ or $\bar{\lambda}$ is among the labels appearing in $\tilde{a}$.

3.2 Stuck-Freedom, Ready Refusal, and Conformance

If two processes, P and Q, communicate via local names in $\tilde{a}$, we can write the system as $(\nu\tilde{a})(P|Q)$. Since $\tilde{a}$ are names local to P and Q, no further environment can interact on $\tilde{a}$, and therefore it makes sense to test whether the interaction of P and Q on $\tilde{a}$ succeeds completely. Informally, we call a process *stuck* on $\tilde{a}$ if it cannot make any progress, and some part of it is ready to communicate on a name in $\tilde{a}$: the communication on $\tilde{a}$ has not succeeded, because it did not finish. In our applications, P is typically a model of a program that has a local connection $\tilde{a}$ to a process whose specification (contract) is Q. We wish to check that the interaction between P and the implementation represented by Q is *stuck-free*, i.e., it cannot get stuck. Stuck-freedom is a safety property and can be checked by a reachability analysis of the system $P \mid Q$. The following definitions make the notions of stuckness and stuck-freedom precise.

Definition 3 (Stuck process). *A process P is called* stuck on $\tilde{a}$, *if $(\nu\tilde{a})P$ is an end-state, and $P \xrightarrow{\lambda}$ for some $\lambda \bar{\in} \tilde{a}$. We refer to such λ as a residual action.*

Definition 4 (Stuck-free processes). *A process P is called* stuck-free on $\tilde{a}$ *if there is no P' and $\tilde{\alpha}$ such that $P \xrightarrow{\tilde{\alpha}} P'$ with $\tilde{\alpha} \cap \tilde{a} = \emptyset$, and P' is stuck on $\tilde{a}$.*

In the situation mentioned above we apply Definition 4 by searching for a transition $P \mid Q \xrightarrow{\tilde{\alpha}} P' \mid Q'$ such that $\tilde{\alpha} \cap \tilde{a} = \emptyset$ and $P' \mid Q'$ is stuck on $\tilde{a}$. The restriction $(\nu\tilde{a})$ in Definition 3 and the condition $\tilde{\alpha} \cap \tilde{a} = \emptyset$ in Definition 4 enforce that only internal reactions (τ-actions) can happen on names in $\tilde{a}$, consistently with $\tilde{a}$ being local. Since $P' \mid Q' \xrightarrow{\lambda}$ for some $\lambda \bar{\in} \tilde{a}$, the interaction between P and Q ends with a residual action, λ, that cannot be matched by any co-action, $\bar{\lambda}$. If λ models an input action, then a component is waiting to receive a message that never arrives, and if λ models a send action, then a component is sending a message that is never consumed. An example of the latter is a remote exception message that is not handled. Such situations are important indicators of problems in many asynchronous applications. Section 4 has examples of how we have used Definition 4 in checking conformance of actual systems.

We seek a conformance relation, $\leq$, such that $S \leq Q$ guarantees the substitutability property, that if $P \mid Q$ is stuck-free, then $P \mid S$ is stuck-free, on any selected names $\tilde{a}$. In the example scenario, this means that it is *safe* to check stuck-freedom of the system $P \mid Q$, where the implementation S is represented by its contract Q. To achieve this goal, the relation $\leq$ must be such that, if $S \leq Q$, then Q gets stuck on $\tilde{a}$ in any context at least as often as S does. This requirement implies that Q may not promise to offer actions that are not actually delivered by S and can be elegantly captured by the notion of refusal in CSP [6]. However, it turns out that the refusal requirement alone is not sufficient, because it does not rule out all cases of residual actions. This motivates the following definitions, where refusals are strengthened to *ready refusals*.

Let $\mathit{init}(P) = \{\alpha \mid P \xrightarrow{\alpha}\}$, and let $\mathcal{L}^{(1)}$ denote the singleton sets of $\mathcal{L}$ together with the empty set, $\mathcal{L}^{(1)} = \{\{\lambda\} \mid \lambda \in \mathcal{L}\} \cup \{\emptyset\}$.

Definition 5 (Refusal). *If X is a subset of $\mathcal{L}$, we say that P* refuses *X if and only if P is stable and* $\mathrm{init}(P) \cap \bar{X} = \emptyset$. *We say that P* can refuse *X if and only if there exists P' such that $P \xrightarrow{\tau^*} P'$ and P'* refuses *X.*

Definition 6 (Readiness). *If $Y \in \mathcal{L}^{(1)}$, we say that P is* ready on *Y, if and only if P is stable and $\lambda \in Y$ implies $P \xrightarrow{\lambda}$. Notice that any stable process is trivially ready on $\emptyset$.*

Definition 7 (Ready Refusal). *If $X \subseteq \mathcal{L}$ and $Y \in \mathcal{L}^{(1)}$, we say that P* can refuse *X* while ready on *Y if and only if P can refuse X from a state that is ready on Y, i.e., there exists P' such that $P \xrightarrow{\tau^*} P'$, P' refuses X, and P' is ready on Y.*

Notice that ready sets are defined in terms of actions (Definition 6) but refusals are defined in terms of co-actions (Definition 5). Initially, this can be a bit confusing. For example, the process a refuses $\{a\}$ and is ready on $\{a\}$.

Definition 8 (Conformance Relation). *A binary relation $\mathcal{R}$ on processes is called a* conformance relation *if and only if, whenever $P \mathrel{\mathcal{R}} Q$, then the following conditions hold:*

C1. If $P \xrightarrow{\tau^*\lambda} P'$ then there exists Q' such that $Q \xrightarrow{\tau^*\lambda} Q'$ and $P' \mathrel{\mathcal{R}} Q'$.
C2. If P can refuse X while ready on Y, then Q can refuse X while ready on Y.

Condition [C2] in Definition 8 may appear surprising. It is further motivated and discussed below, including Section 3.3 and Section 3.4.
If $\mathcal{R}_1$ and $\mathcal{R}_2$ are binary relations on processes, we define their composition, denoted $\mathcal{R}_1 \circ \mathcal{R}_2$, by

$$\mathcal{R}_1 \circ \mathcal{R}_2 = \{(P, Q) \mid \exists R.\ (P, R) \in \mathcal{R}_1 \ \wedge\ (R, Q) \in \mathcal{R}_2\}$$

Lemma 1. *Let $\{\mathcal{R}_i\}_{i \in I}$ be a family of conformance relations. Then*

1. *The relation $\cup_{i \in I} \mathcal{R}_i$ is a conformance relation*
2. *For any $i, j \in I$, the relation $\mathcal{R}_i \circ \mathcal{R}_j$ is a conformance relation*
3. *The identity relation on processes is a conformance relation*

Lemma 1.1 shows that we can define $\leq$ as the largest conformance relation by taking the union of all conformance relations, in the way that is standard for CCS bisimulation and simulation [8].

Definition 9 (Conformance, $\leq$). *The largest conformance relation is referred to as* conformance *and is denoted $\leq$. We write $P \leq Q$ for $(P, Q) \in \leq$, and we say that P* conforms to *Q.*

Condition [C2] of Definition 8 ensures that, if $P \leq Q$, then Q gets stuck on a name a as often as P does. In Definition 8 it is very important that the readiness constraint is only imposed one name at a time, by the fact that the ready sets Y are at most singleton sets (choosing $Y = \emptyset$ yields the standard refusals condition with no readiness constraint, as a special case.) This allows a specification to be more nondeterministic than its refinements, because the specification may resolve its nondeterminism differently for each name. For example, we have $a+b \leq a\#b$. Considered over the alphabet of names $\{a, b\}$, the process $a + b$ refuses $\{a, b\}$ and is ready on both $\{a\}$ and on $\{b\}$. The process $a\#b$ refuses $\{a, b, \bar{b}\}$ from the state a, and it refuses $\{a, \bar{a}, b\}$ from the state b. From the state a, it is ready on $\{a\}$, and from the state b it is ready on $\{b\}$. Therefore, condition [C2] is satisfied. The reader may wish to verify a few more interesting examples:

$$a \mid b \;\leq\; (a.b)\#(b.a), \qquad (a.b)\#(b.a) \;\not\leq\; a \mid b, \qquad a \mid b \;\cong\; a.b + b.a$$

where we write $P \cong Q$ if and only if $P \leq Q$ and $Q \leq P$. We also remark that $\equiv \; \subseteq \; \leq$, by rule [CONG] (further algebraic properties of conformance can be found in [5].)

Proposition 1. *The conformance relation $\leq$ is reflexive and transitive.*

The following theorems state our main theoretical results, precongruence (the operators of CCS are monotonic with respect to conformance, Theorem 1) and substitutability (specifications can safely be substituted for processes that conform to them, Theorem 2). Full proofs are given in [5]. Let C range over CCS contexts, which are process expressions with a "hole" (written []) in them:

$$C ::= [] \mid (P \mid []) \mid ([] \mid P) \mid (\alpha.[] + M) \mid ((\nu a)\ [])$$

We write $C[Q]$ to denote the process expression that arises by substituting Q for the hole in C.

Theorem 1 (Precongruence). *$P \leq Q$ implies $C[P] \leq C[Q]$.*

Theorem 2 (Substitutability). *Assume $P \leq Q$. Then $C[Q]$ stuck-free on $\tilde{a}$ implies $C[P]$ stuck-free on $\tilde{a}$.*

The following two sections imply that conformance is finer than stable failures refinement in CSP (when conformance is defined in terms of traces or, alternatively, when stable failures refinement is defined in terms of transitions) and coarser than refusal preorder in CCS. In this sense, conformance is in between.

3.3 Conformance and Stable Failures Refinement

Condition [C2] in the definition of conformance is adapted from the theory of stable failures refinement in CSP [6, 13], which is based on the concepts of traces and refusals. In the stable failures model, a process P is represented by the set of its *failures*. A failure of P is a pair $(\tilde{\lambda}, X)$ where $\tilde{\lambda}$ is a finite trace of P and X is *refusal set*, i.e., a set of events P can refuse from a stable state after $\tilde{\lambda}$. Process P failure-refines process Q if the traces of P are contained in the traces of Q and the failures of P are contained in the failures of Q.

In our conformance definition we use refusals, but using the stronger ready refusals in condition [C2]. This is motivated by the requirement that conformance should be substitutable with respect to stuck-freedom. As an example, consider the two processes P and Q defined by

$$P = a.\mathbf{0} \quad \text{and} \quad Q = a.\mathbf{0} + \tau.\mathbf{0}$$

Considered as processes over the label set $\{a, \bar{a}\}$, the failures[3] of P are: $\{(\langle\rangle, \{a\}), (\langle a\rangle, \{a, \bar{a}\})\}$, and the failures of Q are: $\{(\langle\rangle, \{a, \bar{a}\}), (\langle a\rangle, \{a, \bar{a}\})\}$. It follows that P failure-refines Q. However, P is stuck on a, but Q cannot get stuck, since its only stable derivative is $\mathbf{0}$, which is not a stuck process. Hence, even though P failure-refines Q, stuck-freedom of Q does not imply stuck-freedom of P. In contrast, the readiness constraint in condition [C2] in our definition of conformance entails that $P \not\leq Q$, because P can perform action a from a stable state (P itself), whereas Q can only perform action a from an unstable state (Q itself).

The natural trace model corresponding to stuck-free conformance modifies the stable failures model to use ready refusals in its failures, so that a failure is now a pair $(\tilde{\lambda}, R)$ where $R = (X, Y)$ is a ready refusal consisting of a refusal set $X \subseteq \mathcal{L}$ and a ready set $Y \in \mathcal{L}^{(1)}$.

The differences between stable failures refinement in CSP and stuck-free conformance results from differences between stuckness and CSP deadlock. In CSP, deadlock is indicated by the absence of a special, successful termination event, $\surd$. Stuckness is directly observable on states in a labeled transition system and is more discriminative than CSP deadlock.

3.4 Conformance and Refusal Preorder

Our notion of conformance bears resemblance to the *refusal preorder* as defined in Iain Phillips' theory of refusal testing [11]. In the refusal preorder, we can observe what happens after a refusal (or a deadlock) by turning refusal sets into observations. Following [2], we can define refusal preorder by adding the transition rule

$$P \xrightarrow{X} P, \quad \text{provided } P \text{ refuses } X$$

for $X \subseteq \mathcal{L}$. For $\rho \in \wp(\mathcal{L}) \cup \mathcal{L}$, we write $P \overset{\rho}{\Longrightarrow} P'$ if and only if $P \xrightarrow{\tau^*} P_1 \xrightarrow{\rho} P_2 \xrightarrow{\tau^*} P'$. We lift to vectors $\tilde{\rho}$ in the usual way. We then define failure traces,

[3] We express failures in terms of maximal refusal sets and use $\langle \ldots \rangle$ for sequences.

f-traces$(P) = \{\tilde{\rho} \in (\wp(\mathcal{L}) \cup \mathcal{L})^* \mid P \overset{\tilde{\rho}}{\Longrightarrow}\}$. The refusal preorder, denoted $\leq_{rf}$, is defined by setting $P \leq_{rf} Q$ if and only if *f-traces*$(P) \subseteq$ *f-traces*(Q). This definition captures simultaneous refusal- and readiness properties. For example, with $\mathcal{L} = \{a, \bar{a}\}$, we have $a \xrightarrow{\{a\}} a \xrightarrow{a} \mathbf{0}$ which shows that a is a stable state from which the set $\{a\}$ is refused and the label a is offered. It is interesting to observe that we have $a \not\leq_{rf} a + \tau.\mathbf{0}$, because the transition $a \xrightarrow{\{a\}} a \xrightarrow{a} \mathbf{0}$ cannot be matched by $a + \tau.\mathbf{0}$. Indeed, the refusal preorder is strong enough to guarantee substitutability for stuck-freedom. However, it is more restrictive than conformance. Consider the processes P and Q defined by

$$P = (a + b.c)\#(c + b.a)$$
$$Q = (a + b.a)\#(c + b.c)$$

It can be checked from the definitions (or see details in [5]) that $P \leq Q$ but $P \not\leq_{rf} Q$. The reason we have $P \leq Q$ is that Q is allowed to choose *different* derivatives as witnesses of condition [C1] and [C2]. Condition [C1] is witnessed by the derivative $(c + b.c)$ of Q, and [C2] is witnessed by the derivative $(a + b.a)$ of Q.

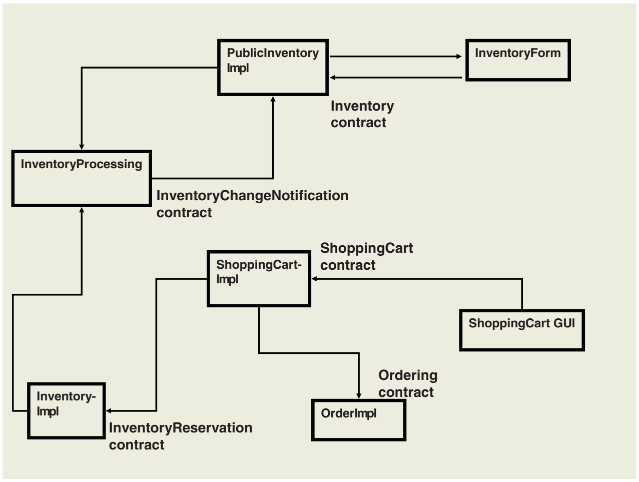
4 Application

Based on the theory presented in Section 3, we have implemented a conformance checker that can be used to analyze asynchronous programs written in common programming languages. Such languages (e.g., C, Java, C#) have several features (such as procedure calls, pointers, shared memory, exceptions, and objects) that are cumbersome to model directly in CCS. Our conformance checker accepts a language ZING, which supports such features directly without the need for complicated encodings. The operational semantics of ZING is given as a labeled transition system. All statements except the send and receive statements are modeled as transitions labeled with τ actions. Transitions from send and receive statements are labeled with the corresponding channel names (ignoring the data values that are sent or received). Note that Definition 8 of conformance makes no reference to the syntax of CCS, and is purely based on the semantics of the labeled transition system. Thus we can define conformance between two ZING processes using Definition 8 directly.

We have applied our conformance checker to check contracts of distributed programs. We next describe how the conformance checks were done and the errors that we found.

Our example application is a distributed program for managing inventory and ordering processes for a bookstore. Contracts are specified as communicating state machines. We used the ZING conformance checker to check that implementations conform to their contract specifications and that implementations do not get stuck with contracts specifying other components that they use.

The structure of the system is shown in Figure 1 (top). It contains five components, `ShoppingCartImpl`, `InventoryImpl`, `OrderImpl`, `PublicInventoryImpl`



```
Contract ShoppingCart =
   AddItem?. ShoppingCart
 + RemoveItem?. ShoppingCart
 + CheckOut?. AcknowledgeOrder!
 + Cancel?. AcknowledgeOrder!


Contract InventoryReservation =
     ReserveInventory?. InventoryReservation
  +  UnreserveInventory?. InventoryReservation
  +  CommitReservation?. Acknowledgement!
  +  CancelReservation?. Acknowledgement!
  +  Timeout
```

Fig. 1. Structure of the Bookstore System (top) and two contracts (bottom)

and `InventoryProcessing`, and, in addition, two user interface processes. There is an arrow from component A to component B, if A *uses* (is a client of) B. Each of the five components has an associated contract, which is a specification of the publically visible behavior of the component. In Figure 1 (top) each contract is named in association with the component it specifies. Two of the contracts are shown in Figure 1 (bottom). A contract specifies a communicating state machine over a set of message types. For example, the `Shoppingcart` contract specifies that the component will repeatedly input messages of type `AddItem` or `RemoveItem` until either a `CheckOut` message or a `Cancel` message is received (we use the notation `T?` for input and `T!` for output, where `T` is a message type.) The `Timeout` event used in the `InventoryReservation` contract is a special event. Timeouts can be attached to input constructs in the implementation language.

The implementation code for a component is placed in a declaration that states the contract it is supposed to implement, as in:

$$\texttt{ShoppingCartImpl} \leq \texttt{ShoppingCart}$$

By relying on the contracts it is possible to automatically translate an implementation into a ZING model that *substitutes contracts for the components it uses.* In doing so, we rely on Theorem 2.

In addition to checking conformance between a component and its contract, we always perform a *stuckness test* that checks for stuck states in the interaction between a component and the contracts it uses. This is necessary, since, for example, a component might have a trivial contract and yet get stuck with components it uses. Such cases are caught and flagged by our conformance checker as well.

The conformance analysis of the `Bookstore` system was done by compiling each component implementation and its associated contract into ZING, resulting in two ZING processes, which were then compared using our conformance checker. The conformance check was done compositionally, by checking one component implementation at a time against its contract specification, substituting contracts to represent the components that are *used*. The entire process is fully automatic. Our conformance analysis detected a number of serious errors including a deadlock, which are briefly described below. In the list below we indicate which contract was checked, and which condition ([C1] or [C2]) of Definition 8 was violated; we also indicate failure of the stuckness test referred to above. We emphasize that these errors were not found before we checked the code[4].

(1) `InventoryReservation`. *Missing timeout specification ([C2]).* The contract failed to specify that the component could timeout on its event loop. This resulted in a ready refusals failure identified by the conformance checker, because the component would refuse all requests after timing out. The contract was corrected by adding a `Timeout` case. A `Timeout` event is modeled as an internal (τ) action, and when attached to an input a it is represented by ZING as $a + \tau.\mathbf{0}$ (assuming here that nothing happens after the timeout).

(2) `InventoryReservation`. *Repeated input not specified ([C1]).* After correction of the previous error, the conformance checker reported a simulation failure. The problem was that the contract specified that the component takes message `CommitReservation` only once, terminating the event loop of the component. However, the implementation did not terminate its loop on receipt of `CommitReservation` as it should, and the implementation was corrected.

(3) `ShoppingCart`. *Stuckness.* Figure 1 shows the contract `ShoppingCart` and the corrected `InventoryReservation`. The implementation uses the components `InventoryImpl` and `OrderImpl`. The conformance check finds that the implementation gets stuck with the `InventoryReservation` contract in a situation where the implementation receives a `CheckOut` message, then goes on to send a `CommitReservation` message to the `InventoryImpl` and then waits to receive an `Acknowledgement`. This receive deadlocks, if the `InventoryImpl` has timed out. In other situations, the timeout causes stuckness by residual messages sent from the `ShoppingCartImpl`. The implementation was corrected.

[4] The code had not been thoroughly tested. Notice, however, that several of the errors would typically not be found by testing.

(4) `Inventory`. *Input not implemented in component ([C2]).* An input specified in the contract `Inventory` but not implemented resulted in a refusals failure. The implementation was corrected.

(5) `InventoryChangeNotification`. *Inputs unavailable after receipt of a particular message ([C2]).* In one particular state, on receipt of message `Done` in its event loop, the imlmentation `PublicInventoryImpl` would exit its event loop without terminating its communication with a publish-subscribe system in the `InventoryProcessing` component, whereas the contract does not reflect this possibility. Messages for that system could therefore be lost, and the implementation was corrected.

5 Conclusion

We have presented a novel refinement relation for CCS and demonstrated that it is suitable for compositional checking of stuck-freedom of communicating processes. Conformance has the advantage of being directly applicable to a labeled transition system in which visible actions and stability can be observed. Stuckness is more discriminative than CSP deadlock by taking orphan messages into account, which is useful for checking asynchronous processes. We proved that conformance is a precongruence on CCS processes satisfying substitutability, and we distinguished it from related process preorders. We have built a conformance checker for ZING, an expressive modeling language, and have applied it to a distributed system.

Acknowledgments

We are grateful to Robin Milner for encouraging comments on a draft of this paper, and to Ranko Lazic for an inspiring conversation. We thank the participants of workshops arranged by Tony Hoare at Microsoft Research, Cambridge in June 2002 (Ernie Cohen, Paul Gardiner, Andy Gordon, Robin Milner, and Bill Roscoe) and at the Computing Laboratory, Oxford University, in November 2003 (Christie Bolton, Michael Goldsmith, Ranko Lazic, and Gavin Lowe). We also wish to thank Tony Andrews, Microsoft, for his work on the design and implementation of ZING.

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.
2. E. Brinksma, L. Heerink, and J. Tretmans. Developments in testing transition systems. In *Testing of Communicating Systems*, IFIP TC6 10th International Workshop on Testing of Communicating Systems, pages 143 – 166. Chapman & Hall, 1997.

3. S.D. Brookes, C.A.R. Hoare, and A.W. Roscoe. A theory of communicating sequential processes. *Journal of the ACM*, 31(3):560–599, 1984.
4. 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.
5. C. Fournet, C.A.R. Hoare, S.K. Rajamani, and J. Rehof. Stuck-free conformance theory for CCS. Technical report, Microsoft Research, 2004.
6. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
7. K.G. Larsen and A. Skou. Bisimulation through probabilistic testing. In *POPL 89: ACM Principles of Programming Languages*, pages 344–352. ACM, 1989.
8. R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.
9. R. Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, 1999.
10. W. Peng and S. Puroshothaman. Towards dataflow analysis of communicating finite state machines. In *PODC 89*, pages 45–58. ACM, 1989.
11. I. Phillips. Refusal testing. *Theoretical Computer Science*, 50(2):241 – 284, 1987.
12. S. K. Rajamani and J. Rehof. Conformance checking for models of asynchronous message passing software. In *CAV 02: Computer-Aided Verification*, LNCS 2404, pages 166–179. Springer-Verlag, 2002.
13. A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1998.