Symbolic Analysis Tools for CSP

Liyi Li, Elsa Gunter, and William Mansky
{liyili2,egunter,mansky1}@illinois.edu

Department of Computer Science,
University of Illinois at Urbana-Champaign

Abstract. Communicating Sequential Processes (CSP) is a well-known formal language for describing concurrent systems, where transition semantics for it has been given by Brookes, Hoare and Roscoe [1]. In this paper, we present trace refinement model analysis tools based on a generalized transition semantics of CSP, which we call HCSP, that merges the original transition system with ideas from Floyd-Hoare Logic and symbolic computation. This generalized semantics is shown to be sound and complete with respect to the original trace semantics. Traces in our system are symbolic representations of families of traces as given by the original semantics. This more compact representation allows us to expand the original CSP systems to effectively and efficiently model check some CSP programs that are difficult or impossible for other CSP systems to analyze. In particular, our system can handle certain classes of non-deterministic choices as a single transition, while the original semantics would treat each choice separately, possibly leading to large or unbounded case analyses. All work described in this paper has been carried out in the theorem prover Isabelle [2]. This then provides us with a framework for automated and interactive analysis of CSP processes. It also gives us the ability to extract Ocaml code for an HCSP-based simulator directly from Isabelle. Based on the HCSP semantics and traditional trace refinement, we develop an idea of symbolic trace refinement and build a model checker based on it. The model checker was transcribed by hand into Maude [3] as automatic extraction of Maude code is not yet supported by the Isabelle system.

1 Motivation

Communicating Sequential Processes (CSP) is a process algebra to describe the behavior and interactions of concurrent systems. Due to the expressive features of external and internal choice together with the parallel composition in CSP, it has been used practically in industry for specifying and verifying concurrent

Acknowledgments. This material is based upon work supported in part by NASA Contract NNA10DE79C and NSF Grant 0917218. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NASA or NSF.

features of various systems, especially ones combining human operators and automations, such as the medical mediator system in Gunter *et al.* [4], the airline ticket reservation system in Wong and Gibbons' paper [5] and interactive systems with human error tolerance in Wright *et al.* [6].

In the traditional semantics of CSP, processes are given semantics via the set of traces they may generate, the set of sequences of individual actions the processes may execute. For example, the CSP process $c?x : B \rightarrow P$ is generally modeled as receiving a single value from the set $\{x|B\}$ and proceeding as P with that value. The set of possible traces will depend on the size of that set. Previous CSP simulators and model checkers have followed this semantics by enumerating all traces individually. In practice, if the set $\{x|B\}$ is an infinite set, current CSP simulators and model checkers actually create an endless number of similar processes and wait for other parts of the program to stop these processes. This affects the efficiency and decreases the scope of analyzable problems for these tools, particularly for model checking.

In this paper, we present a simulator HSim to effectively generate the behaviors of CSP programs, and a model checking tool HMC to check trace refinement properties of CSP programs based on Holistic CSP (HCSP) semantics, a new semantics for CSP processes that uses a symbolic representation of actions to capture a group of properties simultaneously instead of considering only a single element with a single property. The approach we take in this work is to represent families of transitions in CSP by a single transition in HCSP. This allows us to view a set of actions as a whole in some contexts, but also divide it based on various properties in other contexts.

$A.\ (\bigsqcap_{x:x>0\wedge x<10} \mathsf{A}.x \rightarrow \mathsf{SKIP}) [\![\{k.x|k = A \wedge x > 0 \wedge x < 100\}]\!] (\mathsf{A}?x : x > 0 \wedge x < 150 \rightarrow \mathsf{SKIP})$

$B.\ (\bigsqcap_{x:x>0\wedge x<10^5} \mathsf{A}.x \rightarrow \mathsf{SKIP}) [\![\{k.x|k = A \wedge x > 0 \wedge x < 10^6\}]\!] (\mathsf{A}?x : x > 0 \wedge x < 15 * 10^5 \rightarrow \mathsf{SKIP})$

$C.\ (\bigsqcap_{x:x>0\wedge x<10} \mathsf{A}.x \rightarrow \mathsf{SKIP}) [\![\{k.x|k = A \wedge x > 0 \wedge x < 100\}]\!] (\Box_{y:y=1-1} \mathsf{A}?x : x > y \rightarrow \mathsf{SKIP})$

$D.\ (\bigsqcap_{x:x>0\wedge x<10} \mathsf{A}.x \rightarrow \mathsf{SKIP}) [\![\{k.x|k = A \wedge x > 0 \wedge x < 100\}]\!] (\mathsf{A}?x : x > 0 \wedge x < 100 \rightarrow \mathsf{SKIP}) \Box (\mathsf{A}?x : x > 99 \rightarrow \mathsf{SKIP})$

Fig. 1. Example

The differences between the original CSP semantics based tools and the HCSP semantics based tools can be demonstrated by some very simple examples. Four such examples are shown in Figure 1. For each process, the problem was posed, does the process refine itself. The model checker FDR2 [7] can handle case A easily, but it fails to terminate on cases B and C. On process B it begins to run, but eventually generates a stack overflow. When process C is directly input into the CSPM-based simulator ProBE, the whole program crashes. However, the HCSP model checker HMC, we have derived from the semantics given in this paper, easily verifies the trace refinement properties of the processes A - D with respect to themselves in the same amount of time. We will adjust the details of the experiments in Section 5, but from these examples, we can clearly see that the running time of FDR2 depends on the size of the sets bounding choice and parallel composition in each process.

These facts reflect, in part, that the original CSP and Machine-Readable CSP (CSPM) semantics view replicated operators (Replicated Internal Choice, Replicated External Choice, etc) as macros of their binary versions over sets. This means that the original CSP and CSPM syntax cannot express a replicated operator if the set of the replicated operator is infinite, such as the second Replicated Internal Choice operator in process C. Even if the set is finite, the cost is very expensive for CSP-semantics-based tools to run a small replicated process in a large macro, such as in the process B. On the other hand, HCSP-semantics-based tools can overcome this problem and run CSP processes regardless of the size of sets bounding replicated operators. By using HCSP semantics, the three processes in the example will have the same number of possible next moves. This property allows the HCSP-semantics-based tools to run faster than the traditional CSP-semantics-based tools in some cases. In addition, HCSP-semantics-based tools can expand the set of possible CSP processes to analyze, processes B and C above are examples of this fact. We will see that this fact is useful in some real applications such as the medical mediator system in Gunter *et al.* [4].

This paper's contribution is a new formalization of CSP, which we have proved in Isabelle to be consistent with the original semantics. Consequently, we have an HCSP simulator HSim that is directly extracted from the HCSP semantics in Isabelle. Based on this semantics, we introduce symbolic trace refinement which we prove to be equivalent to traditional trace refinement. In addition, we present an HCSP model checker HMC designed to verify the symbolic trace refinement property for HCSP programs. We show families of examples that our HCSP-semantics tools can analyze quite efficiently that previous tools could not analyze. But the symbolic semantics and trace refinement has additional benefits. Firstly, the data set in CSP choice operators can be calculated at runtime, allowing us to model check some processes similar to process C in Figure 1. The HMC is able to decide the value of variable y at runtime and determine the set $\{x|x > y\}$ by using the value of y. Secondly, HCSP is designed to model the broadcast message passing mechanism, allowing us to define point-to-group communication. This mechanism is particularly useful in describing human interactive communication with computers. One such example is the medical mediator system in Gunter *et al.* [4]. Lastly, we show how to define our symbolic trace refinement relation as a proof system based on the HCSP semantics. This proof system makes explicit the relationship between our definition of symbolic trace refinement and the underlying algorithm for HMC, allowing us to easily verify the the two are equivalent. Thus, we can essentially understand the HMC algorithm as an implementation of each rule of the symbolic trace refinement proof system.

2 Syntax and Semantics

The syntax of HCSP and informal meaning is given in Figure 2. For the remainder of this paper, the following name conventions will be used. We will use P and Q for processes. Lower case p refers to an HCSP process name. The letter c

represents an HCSP channel, while the letter a represents an HCSP action. The letter B is a proposition describing the property of a set. In HCSP, we include both *variables* and *parameters*, which are distinct types. We use k for variables ranging over HCSP channels and x for variables over actions. We use U and V to refer to parameters ranging over channels and actions. Variables and parameters serve similar functions, but differ as follows: variables may occur free or bound in HCSP processes and may be replaced by actions or channels by substitution, while parameters occur essentially as local constants not subject to binding or substitution. In the rest of the paper, we will use *freeParams* to refer to a function returning all free parameters in an expression of arbitrary type. We will use l to represent a transition label. Finally, the Greek letter ρ refers to an assignment function that assigns values to parameters. To facilitate the application of HCSP to specific examples, it is parameterized by four user-defined types: a type of expressions for actions and channels (acts), a type of propositions, a type of process names and a type of values to be assigned to acts. One remark must be made here concerning the scope of variables. In the processes $c?x : B \rightarrow P$, $\bigsqcap\limits_{x:B} P$ and $\mathop{\Box}\limits_{x:B} P$, the scope of variable x is both the proposition B and the process P, while the scope of the variables k and x is only the proposition B in the processes $P[\![\{k.x|B\}]\!]Q$ and $P \setminus \{k.x|B\}$.

$P =$

	Ω	Successful termination	|	SKIP	Awaiting successful termination
|	STOP	Unexpected termination	|	$c.a \rightarrow P$	Prefix by action a on channel c
|	$\mathsf{if}\ b\ \mathsf{then}\ P\ \mathsf{else}\ Q$	If statement	|	$\$p$	Process name p as prcoess
|	$P;Q$	Sequential execution	|	$P \Box Q$	Binary external choice
|	$P \sqcap Q$	Binary internal choice	|	$\bigsqcap\limits_{x:B} P$	Replicated internal choice
|	$c?x : B \rightarrow P$	External set prefix			
|	$P[\![\{k.x \mid B\}]\!]Q$	Parallel composition	|	$\mathop{\Box}\limits_{x:B} P$	Replicated external choice
|	$\mathsf{let}\ p = P\ \mathsf{in}\ Q$	Local process name binding			
			|	$P \setminus \{k.x \mid B\}$	Hiding over a set of actions

Fig. 2. HCSP Syntax

The syntax of HCSP differs from that of CSPM by Bryan Scattergood [8] in three ways. Firstly, the actions of CSP are explicitly divided into channels and actions (written $c.a$) in HCSP syntax. Secondly, for the sets used in constructs such as the parallel composition of two processes or replicated internal choice, we use a set comprehension notation. This decomposition of sets into variables and predicates will facilitate the statement of the transition rules of HCSP semantics. Finally, HCSP currently lacks the CSP Renaming operator.

Representative rules for the semantics for HCSP is given in Figures 3 - 4. The full semantics may be found in the technical report of HCSP [9]. The semantics is a merge of the original CSP transition semantics given by Brookes *et al.* [10] with ideas from Floyd-Hoare Logic [11] and symbolic computation. In order to describe the HCSP semantics, there are some functions that need to be supplied for the evaluations of user-defined types. A family of substitution functions $T[a/x]$ is needed for the replacement of variables by acts in each of acts, propositions, and process names. Using these, we define the substitution function for processes. There also needs to be a family of user-defined evaluation functions for acts and a "models" function, $\models$, for checking whether a proposition is true under a given assignment function. We define the functions $sem(\rho, P)$ and $sem(\rho, l)$ as interpretation functions to interpret a given HCSP process or label as a CSP process or label with valuation ρ. The labels of the HCSP semantics will be ranged over by l as follows:

$$l = \surd \mid \tau \mid (U.V)$$

The label $\surd$ represents process completion, the label τ represents a process performing an invisible action, and the label $(U.V)$ represents a pair of parameters, one for a channel and one for a real action. In any execution of a process in accordance with this semantics, the sequence of transitions is labeled with mutually distinct pairs of parameters $(U.V)$, when not labeled by $\surd$ or τ.

$$(\alpha, \gamma, P \sqcap Q) \xrightarrow{\tau} (\alpha, \gamma, P) \quad \textsf{Int\_choice1} \qquad (\alpha, \gamma, P \sqcap Q) \xrightarrow{\tau} (\alpha, \gamma, Q) \quad \textsf{Int\_choice2}$$

$$\frac{(\alpha, \gamma, S, P) \xrightarrow{(U.V)} (\alpha', \gamma', S', P') \quad \exists \rho \,.\, \rho \models (\neg B[U/k][V/x] \wedge \gamma')}{(\alpha, \gamma, S, P [\![\{ k.x | B \}]\!] Q) \xrightarrow{(U.V)} (\alpha', \neg B[U/k][V/x] \wedge \gamma', S', P' [\![\{ k.x | B \}]\!] Q)} \ \textsf{Par\_out1}$$

$$\frac{(\alpha, \gamma, S, Q) \xrightarrow{(U.V)} (\alpha', \gamma', S', Q') \quad \exists \rho \,.\, \rho \models (\neg B[U/k][V/x] \wedge \gamma')}{(\alpha, \gamma, S, P [\![\{ k.x | B \}]\!] Q) \xrightarrow{(U.V)} (\alpha', \neg B[U/k][V/x] \wedge \gamma', S', P [\![\{ k.x | B \}]\!] Q')} \ \textsf{Par\_out2}$$

$$\frac{\begin{array}{c} \exists \rho \,.\, \rho \models (U = U' \wedge V = V' \wedge B[U/k][V/x] \wedge \gamma'') \\ (\alpha, \gamma, S, P) \xrightarrow{(U.V)} (\alpha', \gamma', S', P') \qquad (\alpha', \gamma', S', Q) \xrightarrow{(U'.V')} (\alpha'', \gamma'', S'', Q') \end{array}}{(\alpha, \gamma, S, P [\![\{ k.x | B \}]\!] Q) \xrightarrow{(U.V)} (\alpha'', U = U' \wedge V = V' \wedge B[U/k][V/x] \wedge \gamma'', S'', P' [\![\{ k.x | B \}]\!] Q')} \ \textsf{Par\_in}$$

Fig. 3. HCSP semantics (part of category One and category Two)

We present a labeled transition system for HCSP over quadruples of the form (α, γ, S, P) or (β, ϕ, T, Q), which are called configurations in this paper, where P and Q are HCSP processes, γ and ϕ are environment condition propositions in HCSP that are intended to state the current requirements for parameters "in scope", including those occurring free in P, and α and β are sets of parameters large enough to contain all parameters occurring free in P or γ. The tuples $(l, \alpha, \gamma, S, P)$ or (l, β, ϕ, T, Q) are called moves in this paper. We carry α (or β) with us to allow for the choice of fresh parameter names guaranteed not to clash with a potentially bigger scope than the one locally presented by P (or Q) and γ (or ϕ). S and T are the interpretation functions for interpreting a process name p in a given HCSP process. The environment conditions γ (or ϕ) play the role of providing the pre- and post-condition for each transition. The values potentially represented by labels of the form $(U.V)$ are progressively restricted by the conditions in each of the subsequent quadruples resulting from each transition in the execution. In this way, a single execution in the transition semantics of HCSP potentially represents a parameterized family of executions from the original CSP semantics.

When translating the original CSP semantics into HCSP semantics, the main task is to merge information about actions and channels into the environment condition γ, and use this condition when we are trying to evaluate a CSP process. To do so, we will divide the transition rules in CSP into three categories. The first category contains rules for basic operators having no side conditions other than $\surd$-τ label constraints. The rules without side conditions are those for SKIP, STOP, Internal Choice, External Choice and Sequential Composition operators. The second category contains rules with side conditions needing to be translated into the HCSP framework, i.e., the operators with set or boolean guard information, namely, Parallel, Hiding and If-then-else. The third category includes rules for operators that were treated as macros in CSP. They are all replicated operators, which in CSP are considered to be macros based on other rules. The details of the translation is given in the technical report [9]. The HCSP semantics has been proved sound and complete with respect to the original CSP semantics. We will state only the soundness and relative completeness theorems below. The proof sketches can be found in the technical report [9].

$$\frac{U \notin \alpha \quad V \notin \{U\} \cup \alpha \quad \exists \rho\ .\ \rho \models (U = c \wedge V = a \wedge \gamma)}{(\alpha, \gamma, S, c.a \to P) \xrightarrow{(U.V)} (\{U, V\} \cup \alpha, U = c \wedge V = a \wedge \gamma, S, P)}\ \textsf{Act\_prefix}$$

$$\frac{U \notin \alpha \quad V \notin \{U\} \cup \alpha \quad \exists \rho\ .\ \rho \models (U = c \wedge B[V/x] \wedge \gamma)}{(\alpha, \gamma, S, c?x : B \to P) \xrightarrow{(U.V)} (\{U, V\} \cup \alpha, U = c \wedge B[V/x] \wedge \gamma, S, P[V/x])}\ \textsf{Ext\_prefix}$$

$$\frac{U \notin \alpha \quad \exists \rho\ .\ \rho \models (B[U/x] \wedge \gamma)}{(\alpha, \gamma, S, \bigsqcap_{x:B} P) \xrightarrow{\tau} (\{u\} \cup \alpha, B[U/x] \wedge \gamma, S, P[U/x])}\ \textsf{Rep\_int\_choice}$$

$$\frac{U \notin \alpha \quad (\{u\} \cup \alpha, B[U/x] \wedge \gamma, S, P[U/x]) \xrightarrow{\tau} (\alpha', \gamma', S', P') \quad \exists \rho\ .\ \rho \models \gamma'}{(\alpha, \gamma, S, \Box_{x:B} P) \xrightarrow{\tau} (\alpha', \gamma', S', (\Box_{x:B \wedge x \neq U} P) \Box P')}\ \textsf{Rep\_ext\_tau}$$

$$\frac{U \notin \alpha \quad l \neq \tau \quad (\{u\} \cup \alpha, B[U/x] \wedge \gamma, S, P[U/x]) \xrightarrow{l} (\alpha', \gamma', S', P') \quad \exists \rho\ .\ \rho \models \gamma'}{(\alpha, \gamma, S, \Box_{x:B} P) \xrightarrow{l} (\alpha', \gamma', S', P')}\ \textsf{Rep\_ext\_nor}$$

Fig. 4. HCSP semantics (category Three)

Theorem 1 (Soundness). *For all HCSP processes P, P', assignments ρ, environment conditions γ, γ' and process environments S, S' such that $\rho \models \gamma$ and $\rho \models \gamma'$, and parameter set α such that $freeParams(P) \cup freeParams(\gamma) \subseteq \alpha$ and $(\forall p.p \in \mathtt{dom}(S) \Rightarrow freeParams(S(p)) = \emptyset)$, if $(\alpha, \gamma, S, P) \xrightarrow{l} (\alpha', \gamma', S', P')$, then $sem(\rho, P) \xrightarrow{sem(\rho,l)} sem(\rho, P')$.*

Theorem 2 (Relative Completeness). *Let P be an HCSP process, ρ be an assignment, γ be an environment condition, S be a process environment such that $\rho \models \gamma$, and α be a parameter set such that $freeParams(P) \cup freeParams(\gamma) \subseteq \alpha$ and $(\forall p.p \in \mathtt{dom}(S) \Rightarrow freeParams(S(p)) = \emptyset)$, such that $sem(\rho, P) \xrightarrow{i} T$ in CSP semantics. Then there exist an HCSP process P', an assignment ρ', an environment condition γ', a parameter set α', a process environment S', and a label l such that $i = sem(\rho', l)$, $\rho'|_\alpha = \rho|_\alpha$, $T = sem(\rho', P')$, $\rho' \models \gamma'$ and $(\alpha, \gamma, S, P) \xrightarrow{l} (\alpha', \gamma', S', P')$.*

All the work was done in the interactive theorem prover Isabelle/HOL [2]. Our Isabelle code may be found at `http://www.cs.illinois.edu/~egunter/fms/HCSP/hcsp.tar.gz`.

3 Symbolic Trace Refinement

In order to describe trace refinement model checking in Section 4, we define trace refinement in Definition 1, which is given by Roscoe in [12]. In implementations of CSP trace refinement checker, such as FDR2, they actually check the refinement property by using some relation which is similar to trace simulation. We list it in Definition 2. This definition is more similar to the trace simulation definition. Current trace refinement checkers, such as FDR2, use this approach with memoization to prove the trace refinement between two programs. In Back and Wright's paper [13], they actually prove that this relation implies the trace refinement relation described in Definition 1.

Definition 1 (Trace Refinement). $Q \sqsubseteq_T P = \mathit{trace}(P) \subseteq \mathit{trace}(Q)$.

Definition 2 (Simulation Trace Refinement). *The relation P* trace refines *Q, written $Q \sqsubseteq_S P$, is the smallest relation satisfying the following:*

- $\Omega \sqsubseteq_S \Omega$
- $Q \sqsubseteq_S \mathsf{STOP}$
- *If there exists Q' such that $Q \xrightarrow{\tau} Q'$ and $Q' \sqsubseteq_S P$, then $Q \sqsubseteq_S P$*
- *If for all P' we have $P \xrightarrow{l} P'$ implies*
  - *either $l = \tau$ and $Q \sqsubseteq_S P'$,*
  - *or $l \neq \tau$ and there exists Q' such that $Q \xrightarrow{l} Q'$ and $Q' \sqsubseteq_S P'$,*

  then $Q \sqsubseteq_S P$

Definition 2 only works in traditional CSP semantics. In order to describe the symbolic semantics, we need to be able to connect environment predicates with processes. Hence, we define symbolic trace refinement in Definition 3. This definition is actually more similar to the definition of trace simulation, but we call it symbolic trace refinement here because it is equivalent to the simulation trace refinement definition above. In this definition, the function $\eta : (\phi, \beta, \alpha) \rightarrow (\alpha', \phi')$ is a renaming function to rename all parameters of a given environment predicate ϕ that are in the set β to new parameters that are not in the set $\alpha \cup \beta$, and return the resulting environment condition ϕ' and the new parameter set α' containing all parameters in the set α and the environment condition ϕ'. This function is useful in the final rule of symbolic trace refinement. In each inductive step $(\beta_i, \phi_i, T_i, Q_i) \sqsubseteq_T (\alpha'_i, \gamma' \wedge \phi'_i, S', P')$ for $i \in [1..n]$ in the final rule, we want to bind the environment condition of the implementation configuration γ' by an extra constraint ϕ'_i.

Definition 3 (Symbolic Trace Refinement). *The relation* $(\beta, \phi, T, Q) \sqsubseteq_{ST} (\alpha, \gamma, S, P)$ *is the smallest relation satisfying the following:*

- $(\beta, \phi, T, \Omega) \sqsubseteq_{ST} (\alpha, \gamma, S, \Omega)$
- $(\beta, \phi, T, Q) \sqsubseteq_{ST} (\alpha, \gamma, S, \mathsf{STOP})$
- *If there exists* (β', ϕ', T', Q') *such that* $(\beta, \phi, T, Q) \xrightarrow{\tau} (\beta', \phi', T', Q')$ *and* $(\beta', \phi', T', Q') \sqsubseteq_{ST} (\alpha, \gamma, S, P)$*, then* $(\beta, \phi, T, Q) \sqsubseteq_{ST} (\alpha, \gamma, S, P)$
- *If for all* $(\alpha', \gamma', S', P')$ *we have* $(\alpha, \gamma, S, P) \xrightarrow{l} (\alpha', \gamma', S', P')$ *implies*
  - *either* $l = \tau$ *and* $(\beta, \phi, T, Q) \sqsubseteq_{ST} (\alpha', \gamma', S', P')$,
  - *or* $l = \surd$ *and there exists* (β', ϕ', T', Q') *such that* $(\beta, \phi, T, Q) \xrightarrow{\surd} (\beta', \phi', T', Q')$ *and* $(\beta', \phi', T', Q') \sqsubseteq_{ST} (\alpha', \gamma', S', P')$,
  - *or* $l = (U.V)$ *and there exists a natural number* n*, configurations* $(\beta_i, \phi_i, T_i, Q_i)$*, parameters* U' *and* V'*, parameter sets* α'_i *and environment conditions* ϕ'_i *such that,*
    - ▷ $(\alpha \cup \alpha' \cup \bigcup_{i=1}^{n} \alpha'_i) \cap (\beta \cup \bigcup_{i=1}^{n} \beta_i) = \emptyset$
    - ▷ $\bigwedge_{i=1}^{n}((\beta, \phi, T, Q) \xrightarrow{(U_i.V_i)} (\beta_i, \phi_i, T_i, Q_i))$
    - ▷ $\bigwedge_{i=1}^{n}(\exists \rho.\rho \models \gamma' \wedge \phi_i[U/U_i][V/V_i])$
    - ▷ $(\exists \rho'.\forall c\ a\ .\rho'[U \mapsto c, V \mapsto a] \models \gamma' \Rightarrow \bigvee_{i=1}^{n} \rho'[U_i \mapsto c, V_i \mapsto a] \models \phi_i)$
    - ▷ $\bigwedge_{i=1}^{n}((\alpha'_i, \phi'_i) = \eta(\phi_i[U/U_i][V/V_i], \beta_i, \alpha'))$
    - ▷ $\bigwedge_{i=1}^{n}((\beta_i, \phi_i, T_i, Q_i) \sqsubseteq_{ST} (\alpha'_i, \gamma' \wedge \phi'_i, S', P'))$

  then $(\beta, \phi, T, Q) \sqsubseteq_{ST} (\alpha, \gamma, S, P)$

The symbolic trace refinement definition has the following relation with the original trace refinement definition.

Theorem 3 (Symbolic Trace Refinement Relation). *For all HCSP configurations* (α, γ, S, P) *and* (β, ϕ, T, Q)*, assignments* ρ *and* δ *such that* $\beta \cap \alpha = \emptyset$*,* $\mathit{freeParams}(P) \cup \mathit{freeParams}(\gamma) \subseteq \alpha$*,* $(\forall p.p \in \mathtt{dom}(S) \Rightarrow \mathit{freeParams}(S(p)) = \emptyset)$*,* $\rho \models \gamma$ *and* $\delta \models \phi$*, we have* $(\beta, \phi, T, Q) \sqsubseteq_S T(\alpha, \gamma, S, P)$ *if, and only if* $\mathit{sem}(\delta, Q) \sqsubseteq_T \mathit{sem}(\rho, P)$.

Proof. (*Sketch*) We first do an induction on rules of trace refinement to prove the "only if" side of the theorem. We prove this direction by using relatively completeness described in Theorem 2. We then prove the "if" side of the theorem by induction on rules of symbolic trace refinement with the soundness theorem described in Definition 1 □

4 HCSP Simulator and Model Checker

To put the theory of HCSP into practice, we have specified an HCSP simulator with a rich mutually recursive datatype for actions and propositions in Isabelle. OCaml code for the simulator, which we call HSim, is then extracted from the Isabelle specification directly. The core of HSim is included with the package for the soundness and completeness theorems. In HSim, we have limited the propositions to quantifier-free first order logic with Presburger arithmetic in order to maintain decidability. In doing so, we render the single-step transition relation computable as a function generating a finite set of possibilities. We then represent all possible traces with a lazy stream data structure supporting back-tracking. This enables us to incrementally compute the requirements for a given trace, which can be inspected at each step, and can be back-tracked when the requirements are proven to be unsatisfiable. Using the HCSP semantics, we can indefinitely delay the calculation of a specific trace using trace patterns and pre- and post-conditions, until one trace pattern / condition is selected. At this point, satisfiability analysis can be used to generate an instance trace, if such is desired.

In the case of the medical mediator example, we were able to use the simulator specification in Isabelle to enumerate the possible trace patterns for the System, and to verify that all traces satisfying each pattern-condition so enumerated satisfy a pattern-condition of the Safety process.

In addition to HSim, we have implemented in the rewriting logic engine Maude [3] a model checker HMC to check the trace refinement property of HCSP programs. This implementation is a hand transcription of the Isabelle specification of HMC. Previous model checkers for CSP, including FDR2 and PAT, check trace refinement between two processes by explicit trace enumeration with circularity checking. In contrast, as we did with the simulator HSim, the model checker HMC is based on the symbolic semantics and symbolic trace refinement described in Section 2 and 3.

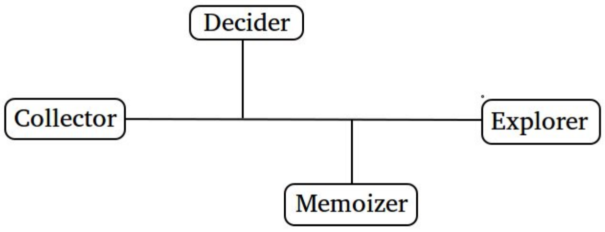

Fig. 5. Algorithm Components

We describe the algorithm of HMC by breaking it into four components: the explorer, the decider, the memoizer and the collector. The explorer is the core of the trace refinement graph searching algorithm. It controls how we do the trace refinement at the meta level regardless of the details of the HCSP semantics. The decider is an auxiliary SMT solver to determine whether a next possible move of a given HCSP configuration is valid based on the satisfiability of the constraint in the next possible move. The memoizer controls how we determine the one configuration is an instance of another configuration that arise during circularity checking, while the collector controls how we collect the next possible moves of a given HCSP configuration.

At the explorer level, our algorithm can be generalized as a four-step procedure based on the definition of symbolic trace refinement described in Section 3. We start with a pair of configurations of the implementation and specification, two τ-labeled memoizing sets (one for the implementation and one for the specification) and a configuration-pair memoizing set for checking a pair of configurations. Step One is to get the next possible moves of a given implementation configuration and a given specification configuration by the collector. Based on these sets of moves, we remove the inconsequential and invalid moves. A move is inconsequential if it is a τ-labeled move that is an instance to one in the implementation or specification τ-memoizing set. An invalid move is one with an unsatisfiable environment condition. In Maude, Step One is implemented by the Maude function verify.

Step Two is to replace all τ-labeled moves of the implementation from the set collected in Step One by the set of non-τ-labeled moves that are reachable by a possibly empty sequence of τ-labeled moves from the τ-labeled move being replaced. Both the τ-labeled move being replaced and all subsequent τ-labeled moves transitioned over in a sequence leading to a replacing non-τ-labeled move are added to the implementation τ-memoizing set, if it is not an instance to one already present. If it is an instance to one already present, we cut that search branch and backtrack to look for other τ-labeled sequences. The result of Step Two is that the all moves collected for the implementation have the label form $U.V$. In Maude, the function verifyTau will handle this step.

Step Three is to check for each move the implementation if there exists a move of the specification that refines the given move of the implementation. It has three operations. The first operation processes the next-moves set for the specification generated in Step One. This operation preforms the same reduction/replacement action on τ-labeled moves of the specification as was done in Step Two for the implementation, except that the replacement is only of a single reachable non-τ move, rather than the full set. All τ-labeled moves in the sequence for the one in the given next-moves set to its replacement are added to the specification τ-memoizing set. This operation is only performed when there do not exist any more $U.V$-labeled moves in the specification next-moves set. We are repeatedly applying this operation until there is a move of the specification with the label form $U.V$, or there are no more moves of the specification. In the latter case, we answer false.

In the second operation, a move of the specification with the label $U'.V'$ and environment condition γ' is selected. For the given move of the implementation with label $U.V$ and environment condition γ, we check whether or not the formula $\forall U.V.\gamma \Rightarrow \gamma'[U/U'][V/V']$ is satisfiable, using the decider also implemented in Maude. Here, we are taking advantage of our symbolic semantics for HSCP. If one such specification move can be found, we can move onto Step Four directly. If not, we will go to the third operation. In the third operation, we assume that we have the given move of the implementation $(U.V, \alpha, \gamma, S, P)$. We check whether there is a set of next moves of the specification $(U'_i.V'_i, \beta_i, \phi_i, T_i, Q_i)$ for $i = 1, \ldots, n$ such that for each i the formula $\gamma \wedge \phi_i[U/U'_i][V/V'_i]$ is satisfiable, and the formula $\forall U\, V.\gamma \Rightarrow \bigvee_{i=1}^{n} \phi_i[U/U'_i][V/V'_i]$ is satisfiable. If there are such moves of the specification, we construct the new parameter sets and new environment conditions as $(\alpha'_i, \phi'_i) = \eta(\phi_i[U/U'_i][V/V'_i], \beta_i, \alpha)$ for $i = 1, \ldots, n$, where η is the renaming function of Section 2. We also construct the new pairs of configurations $((\alpha'_i, \gamma \wedge \phi'_i, S, P), (\beta_i, \phi_i, T_i, Q_i))$ for $i = 1, \ldots, n$. We then allocate Step Four checks for each of these pairs of configurations. If there do not exist such moves of the specification, we will answer false. Step Three is implemented as functions verifyAction and verifyActionAux in Maude.

Step Four checks if the new input pair of implementation-specification configurations from Step Three is an instance of one in the current configuration-pair memoizing set. If it is, then we answer true; otherwise, we go back to Step One to check the new implementation-specification configuration pair with adding the implementation-specification configuration pair into the current configuration-pair memoizing set. In Maude, we implement this step by the function preverify.

The four-step procedure is a general framework and specification for the symbolic graph searching algorithm. There are many ways to implement this procedure. We implement one such transcription in Maude. It is a combination of a breadth-first search algorithm, a depth-first search algorithm and circularity checking.

The memoizer checks when an element is an instance of an element in a given memoizing set. The memoizer can be easily implemented by checking whether an element is an instance of a specific element in a set except that We need to define instance for configurations and extend to configuration pair with covariance in the first component and contravariance in the second.

Definition 4 (Configuration Instance). *We say that* $((\alpha, \gamma, S, P)$ *is an* instance *of* $(\alpha', \gamma', S', P'))$ *and write* $(\alpha, \gamma, S, P) \rightarrow (\alpha', \gamma', S', P')$ *if there exists renaming functions* ζ *for parameters and* ξ *for process variables, such that* $P = \zeta(\xi(Q))$*, and for all* p *as process variable,* $S(p) = \xi(S'(\xi^{-1}(p)))$ *and* $\gamma \Rightarrow \zeta(\gamma')$*. A configuration pair* $((\alpha, \gamma, S, P),\ (\beta, \phi, T, Q))$ *is an* instance *of* $((\alpha', \gamma', S', P'), (\beta', \phi', T', Q'))$ *if* $(\alpha, \gamma, S, P) \rightarrow (\alpha', \gamma', S', P')$ *and* $(\beta', \phi', T', Q') \rightarrow (\beta, \phi, T, Q)$*.*

In the implementation in Maude, rather than using a SMT solver to check the formula $(\gamma \Rightarrow \zeta(\gamma'))$ in Definition 4, we do a weaker syntactic check on the formula. This implementation reduces the heavy use of the SMT solver and might lead to a searching on unnecessary branches, but it will not lead to a false result.

At the collector level, we first implement our HCSP rules in Maude, then we collect all next possible moves of an HCSP configuration by unifying the HCSP configuration with the set of symbolic semantic rules and applying all possible rules on the configuration and put the results in a set. There is a central problem in the implementation of the collector: the structures of an environment condition in an HCSP configuration. we revisit them by borrowing an idea from the

Union Find algorithm and Binary decision diagrams [14] in the implementation of HMC. In the HCSP semantics, we observe that at each move we only add a new constraint conjunctively to the current pre-condition. As a result, when we evaluate an HCSP process long enough, the condition of the environment gets quite large. Since SMT solvers operate in time exponential in the size o fthe input problem, it is critical to keep the size of the problems passed to them as small as possible. The collector serves to accomplish this.

Observation of executing CSP programs indicates that any given conjunct of the environment condition typically shares parameters with only a very few other individual conjuncts. We say two predicates satisfy the *parameter relation* if they share one or more parameters in common. The collector represents the environment condition as a set of sets of individual conjuncts, where each set of individual conjuncts is a connected component of the parameter relation. When a next move is calculated, the new conjunct it contributes to the environment condition is merged with the sets of conjuncts with which it satisfies the parameter relation, forming a new connected component. Since each environment condition is checked for satisfiability, checking the new environment condition can be done by checking satisfiability of just the connected component of the new conjunct contributed by the move. This typically is a much smaller formula to pass to the SMT solver than the entirety of the new environment condition.

After we have confirmed that the connected component of the new conjunct is satisfiable, the collector is also used to reduce the environment condition itself. This is done by calculating the parameters still present in the process of the next move, and removing from the new environment condition all those connected components whose parameters are disjoint from the process parameters. Subsequent moves of a process cannot constraint any previously existing parameters not present in the process. Therefore, given that the current connected components of the environment condition are satisfiable, the satisfiability of subsequent environment conditions is not impacted by the connected components whose parameters are disjoint from the process parameters.

5 Examples and Experiment

$$\mathsf{Clicker}(c,r) = \bigsqcap_{\substack{s:s>0\wedge\\ s\le N}} K.r.c.s \rightarrow \mathsf{Clicker}(c,r) \qquad \mathsf{Broadcast}(r) = \mathop{\Box}_{c:true} K.r.c?s : s>0 \wedge s \le N \rightarrow \mathsf{Out}.r.s \rightarrow \mathsf{Broadcast}(r)$$

$$\mathsf{Room}(r) = (\mathsf{Clicker}(c1_r, r) [\![\{\}]\!] \mathsf{Clicker}(c2_r, r)) [\![\{k.x \mid \exists c\ s.\ \mathsf{hd}(k) = K\}]\!] \mathsf{Broadcast}(r) \qquad \mathsf{Center} = \mathsf{Room}(1) [\![\{\}]\!] \mathsf{Room}(2)$$

$$\mathsf{ATM1} = \mathsf{Incard}?c : (M < c < N) \rightarrow \mathsf{PIN}.c \rightarrow \mathsf{Req}?n : (99 < n) \rightarrow \bigsqcap_{x:x=n\wedge bx<2000} \mathsf{Dispense}.x \rightarrow \mathsf{Outcard}.c \rightarrow \mathsf{ATM1}$$

$$\mathsf{ATM2} = \mathsf{Incard}?c : (M < c < N) \rightarrow \mathsf{PIN}.c \rightarrow \mathsf{Req}?n : (99 < n) \rightarrow \\ \bigsqcap_{x:x=n\wedge bx<2000} \mathsf{Dispense}.x \rightarrow \mathsf{Outcard}.c \rightarrow \mathsf{ATM2} \sqcap (\mathsf{Refuse}.1 \rightarrow \mathsf{ATM2} \sqcap \mathsf{Outcard}.c \rightarrow \mathsf{ATM2})$$

Fig. 6. Examples

Since the HSim keeps track of the constraints of data in an HCSP process, while traditional CSP simulator, such as ProBE, lists values of data in a CSP process. The difference between two kinds of simulators is large enough without doing experiment. In this section, we focus on the experiment between HMC and traditional trace refinement model checker.

Besides the small examples in Fig. 6 and the medical mediator example from Gunter *et al.* [4], there are many other real implementations that can benefit from modeling in the HCSP system. Generally speaking, every real model with several users trying to access one or more copies of a very large database can benefit from the HCSP system. A song broadcasting system and an ATM are two such examples.

Song broadcasting systems are used in entertainment businesses such as discos and karaokes to allow people to select songs from a large database. Such systems typically have a large collection of songs; a collection in excess of 200,000 would not be uncommon. A typical karaoke bar has more than twenty rooms for separate entertainment. Typically, each room has two remote clickers for selecting the next song to be played. After a user selects a song, the remote clicker will send the song selection to the song broadcasting system, which will play it in the room. Since only one song can be broadcast at a time, if two people send selections simultaneously, only one signal will be honored immediately, while the other one will be delayed for later action. We model the karaoke center in CSP in Figure 6. For simplicity, we assume the Karaoke center only has two rooms.

In Fig. 6, the capital letter N refers to an arbitrary number to represent the size of the database that contains all songs in the song broadcasting system. Typically, we know that the number N is a large number, but we do not know exactly how large it is. In order to verify properties in the system, such as safety and deadlock-freedom, it is better to leave the number N to be unspecified. We will set the number N to be 500 and 500,000 as test cases in the experiment.

Likewise, we implement two ATMs in Fig. 6. The two ATM processes are to describe the procedures of a machine that is receiving commands from humans and responding to them. One can easily see that ATM1 can refine ATM2 but not vice versa. We will test positive trace refinement cases of ATM1 to ATM2 as well as negative trace refinement cases of ATM2 to ATM1. In these ATMs, the numbers N and M specify the range of the debit or credit cards that can be read. Typically, a debit or credit card will have sixteen digits. We test the cases when the numbers N and M are one, four and sixteen digits.

We have compared the efficiencies of FDR2 and HMC in some programs. The experiment was run on an Intel core i7 machine with eight gigabytes of memory and a Ubuntu 13.04 system. The testing programs are processes A, B, C and D from Section 1 and the ATMs with N and M being one, four and sixteen digits. We have tested positive trace refinement cases of ATM2 to ATM1 and negative ones of ATM1 to ATM2. In addition, we have tested the Karaoke center examples when N is equal to 500 and 500,000; and the medical mediator examples in which there are two mediators, one nurse, three patients and three devices and when there are two mediators, two nurses, three patients and three devices. The results can be found in Figure 7. From the table, we can see that HMC can finish all the jobs, while FDR2 fails to execute some programs. In most cases, HMC is more efficient at verifying the trace refinement property

Programs	Specifications	FDR2	HMC
Process A	Process A	$<$ 2 secs	$<$ 2 secs
Process B	Process B	N/A	$<$ 2 secs
Process C	Process C	N/A	$<$ 2 secs
Process C	Process D	N/A	$<$ 2 secs
ATM1 one digit	ATM2 one digit	45 secs	$<$ 2 secs
ATM1 four digits	ATM2 four digits	$>$ 12 hours	$<$ 2 secs
ATM1 16 digits	ATM2 16 digits	N/A	$<$ 2 secs
ATM2 one digit	ATM1 one digit	$<$ 2 secs	$<$ 2 sec
ATM2 four digits	ATM1 four digits	$>$ 12 hours	
ATM2 16 digits	ATM1 16 digits	N/A	$<$ 2 secs
Karaoke 5	Karaoke 5	$<$ 2 secs	37 secs
Karaoke 500	Karaoke 500	$>$ 12 hours	37 secs
Karaoke 500,000	Karaoke 500,000	N/A	37 secs
Medical one nurse	Safety one nurse	55 mins	3.6 hours
Medical two nurses	Safety two nurses	N/A	3.8 hours

Fig. 7. Experiment Results

of programs than is FDR2. In addition, FDR2 is very sensitive to the size of the input data, and it cannot recognize the similarity of different programs. It succeeds in model checking some programs, but fails when we change them a little bit. For example, FDR2 can execute process A completely, but fails to even read processes C and D. The medical mediator is a more representative example. Because of the sensitivity of FDR2 with respect to the input data, it can finish the job when there is only one nurse, but not when there are two. On the other hand, even though HMC needs a longer time to finish a job when the input data is small, it can successfully model check the trace refinement property no matter how big the input data gets replicated choice operators.

6 Related Work

Currently, there are several existing CSP simulators and model checkers based on the original CSP transition semantics. CSPM [8] gives a standard CSP syntax and semantics in machine readable form, introduced by Bryan Scattergood, which is based on the transition semantics introduced by Brookes and Roscoe [10]. It provides a standard for many CSP tools, including FDR2 by Formal Systems (Europe) Ltd., the industry standard for CSP model checkers [7]. ProBE [15] is a simulator created by the same group, which simulates a CSP process by listing all the actions and states one by one as a tree structure [15]. Jun Sun *et al.* [16] merged partial order reduction with the trace refinement model checking in the tool called PAT. CSP-Prover is a theorem proving tool built on top of Isabelle [17]. It provides a denotational semantics of CSP in Higher-Order Logic. CSPsim [18] is another simulator based on the CSPM standard. Its major innovation is the use of "lazy evaluation". The basic idea of CSPsim is to keep track of all the current actions, then compare them with the actions of the outside world and only select the possible executable actions for the very next step [18]. The phrase "lazy evaluation" refers to a pre-processing step in which CSPsim selects some processes that contain fewer actions and generates some conditions in advance. After that, CSPsim evaluates the whole program based on these conditions.

These tools use the traditional view of actions as single elements, and so tends to generate a large number of states when comparing multiple possibilities for actions. Additionally they treat some operators, especially replicated operators, as macros, and hence, even though it is possible for some tools (CSP-Prover) to analyze some complicated programs, such as medical mediator, by the theorem proving setting, it is impossible for these tools to generate traces when the replicated set is infinite. The medical mediator project by Gunter *et al.* [4] provides a example of the advantages of HCSP over CSP-semantics based tools. The main goal of the medical mediator project is to prove that the set of traces of the process $System|[Vis]|Given$ is a subset of the traces of $Safety|[Vis]|Given$, where Vis is a set defined as $\{y.(\exists n\ d\ m\ x.\ y = RFIDChan_d^{n,m}.x) \vee (\exists m\ z.\ y = EHRBECh_m.z)\}$. This requires exploring all possible traces generated by the System process. Tools based on the original CSP semantics, such as FDR2, fail when dealing with large or unbounded sets. For example, the Med process as given does not put any restrictions on the sets of values that may be received over various channels. The simulator and model checker we have built based on HCSP semantics benefits from being able to handle such large or unbounded sets uniformly as single actions, thus avoiding state explosion problems.

Along the way we are writting our paper, FDR3 comes out [19]. FDR3 develops a parallelized algorithm for model checking trace refinement property over FDR2. When we use FDR3 to test our programs, the performence is better then the performence of FDR2. However, it still cannot catch the behavior of infinite set in replicated choice operators. For example, when we test the process C and D in Figure 1, FDR3 fails to terminate.

In terms of symbolic semantics, there are several existing symbolic semantics for process algebras, mainly, serving process algebras having similar structure to the Π-calculus. Early work of Hennessy and Lin [20] provides a framework of symbolic semantics for value-passing process algebras. Later, Sangiorgi applied this symbolic semantics idea to the Π-calculus [21]. Bonchi and Montanari revisited the symbolic semantics of the Π-calculus [22], providing a symbolic transition semantics for the Π-calculus by including the predicate environment condition as a part of a label in a transition system. In their symbolic transition semantics, they only discovered the relation in the parallel operator in the Π-calculus.

LOTOS is a kind of process algebras that contains features from both the Π-calculus and CSP. Its parallel operator is similar to that of CSP. The parallel operator contains a middle set to restrict the communication actions between the left and right processes. Calder and Shankland provided a symbolic semantics for LOTOS [23]. As in the work done by Bonchi and Montanari, Calder and Shankland both represented their condition in the label position instead of dividing it into pre- and post-conditions. Pugliese, Tiezzi and Yoshida proposed a symbolic semantics for service-oriented computing COWS that is similar to the Π-calculus [24]. For the works above providing symbolic transition semantics, the condition is placed in the label instead of dividing it into pre- and post-conditions, which makes their symbolic semantics fail to answer differently for different input environments for a same program. In many cases, it is necessary to consider different initial environment conditions and these different conditions lead to different results in CSP programs.

mCRL2 is a process algebra designed to execute symbolically [25]. It is a well-known π-calculus like generic language with symbolic transition semantics to model point-to-point communication. They claimed that people can catch the behavior of infinite set in their choice summation operator. However, the set needs to be determined statically. It means that it cannot catch the behavior similar to process C in Figure 1. In addition, even though mCRL2 claimed the language is generic and we can translate other process algebra into mCRL2, it is very hard for mCRL2 to model a broadcast communication system with point-to-group communication, because they need to know the total number of processes in the universe and send enough messages to each individual process in a group. However, knowing total number of processes is very hard in some cases. For example, if we want to model a group of people who are in a conference reach a consensus at the same time. It is almost impossible for mCRL2 to model this procedure. On the other hand, we can model this procedure easily by using a replicated choice operator to select the total number of people in the conference, then using a replicated parallel operator with consensus value in between to model the fact that all people communicate with each other by the consensus value.

7 Conclusion and Future Work

In this paper we have presented a new semantics for CSP, the HCSP semantics. HSCP provides an alternative way to model CSP processes by viewing transitions as bundles of the original transitions, where all transitions in the bundle can be described by a uniform property derived from the process. By this translation, we can allow HCSP-based tools to run some CSP programs which are not able to run in the original CSP-based tools. We have shown the HCSP semantics to be equivalent to the original CSP transition semantics. We have also presented an HCSP-based simulator, which is extracted directly from the Isabelle code for the HCSP semantics. We show an HCSP-based model checker to check the trace refinement of CSP programs and show that the model checker is very efficient to deal with some CSP programs by experiment. By using several expamles in the experiment, we show that the HCSP semantics based trace refinement model checker can overcome some difficulties that traditional CSP-semantics-based tools cannot handle.

For further study, we are interested in adding semantics to deal with the replicated parallel operators in HCSP and use it in the model checker. We believe that it will significantly increase the efficiency in the model checker to answer trace refinement problem. We also want to generalize our framework to deal with other kinds of transition semantics.

References

1. Brookes, S.D., Hoare, C.A.R., Roscoe, A.W.: A theory of communicating sequential processes. J. ACM **31**(3) (1984) 560–599
2. Paulson, L.C.: Isabelle: The next 700 theorem provers. In Odifreddi, P., ed.: Logic and Computer Science. Academic Press (1990) 361–386
3. Clavel, M., Durn, F., Eker, S., Lincoln, P., Mart-Oliet, N., Meseguer, J., Quesada, J.F.: Maude: Specification and programming in rewriting logic (2001)
4. Gunter, E.L., Yasmeen, A., Gunter, C.A., Nguyen, A.: Specifying and analyzing workflows for automated identification and data capture. In: HICSS, IEEE Computer Society (2009) 1–11
5. Wong, P.Y.H., Gibbons, J.: A process-algebraic approach to workflow specification and refinement. In: Proceedings of the 6th international conference on Software composition. SC'07, Berlin, Heidelberg, Springer-Verlag (2007) 51–65
6. Wright, P., Fields, B., Harrison, M.: Deriving human-error tolerance requirements from tasks. In: In Proc. of ICRE94 IEEE Intl. Conf. on Requirements Engineering, IEEE (1994) 135–142
7. Ltd., F.S.E.: Failures-divergence refinement. FDR2 user manual. In: FDR2 User Manual. (2010)
8. Scattergood, J.: The Semantics and Implementation of Machine-Readable CSP. D.phil., Oxford University Computing Laboratory (1998)
9. Li, L., Gunter, E.L., Mansky, W.: Symbolic semantics for csp. Technical report, Department of Computer Science, University of Illinois at Urbana-Champaign (2013)
10. Brookes, S.D., Roscoe, A.W., Walker, D.J.: An operational semantics for CSP. Technical report, Oxford University Computing Laboratory (1986)
11. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10) (October 1969) 576–580
12. Roscoe, A.W.: The Theory and Practice of Concurrency. Prentice Hall PTR, Upper Saddle River, NJ, USA (1997)
13. Back, R.J.R., Wright, J.V.: Trace refinement of action systems. In: Structured Programming, Springer-Verlag (1994) 367–384
14. Groote, J.F.: Binary decision diagrams for first order predicate logic
15. Ltd, F.S.E.: Process behaviour explorer. ProBE user manual. In: ProBE User Manual. (2003)
16. Sun, J., Liu, Y., Dong, J.S.: Model checking csp revisited: Introducing a process analysis toolkit. In: In ISoLA 2008, Springer (2008) 307–322
17. Isobe, Y., Roggenbach, M.: A complete axiomatic semantics for the CSP stable-failures model. In Baier, C., Hermanns, H., eds.: CONCUR 2006 Concurrency Theory. Volume 4137 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2006) 158–172 10.1007/11817949_11.
18. Brooke, P.J., Paige, R.F.: Lazy exploration and checking of CSP models with CSPsim. In McEwan, A.A., Schneider, S.A., Ifill, W., Welch, P.H., eds.: CPA. Volume 65 of Concurrent Systems Engineering Series., IOS Press (2007) 33–49
19. Gibson-Robinson, T., Philip Armstrong, A.B., Roscoe, A.: Fdr3 - a modern refinement checker for csp. In: TACAS. (2014) In Press.
20. Hennessy, M., Lin, H.: Symbolic bisimulations. Theor. Comput. Sci. **138**(2) (February 1995) 353–389
21. Sangiorgi, D.: A theory of bisimulation for the pi-calculus. In Best, E., ed.: CONCUR. Volume 715 of Lecture Notes in Computer Science., Springer (1993) 127–142

22. Bonchi, F., Montanari, U.: Symbolic semantics revisited. In: Proceedings of the Theory and practice of software, 11th international conference on Foundations of software science and computational structures. FOSSACS'08/ETAPS'08, Berlin, Heidelberg, Springer-Verlag (2008) 395–412
23. Calder, M., Shankland, C.: A symbolic semantics and bisimulation for full lotos. In: PROC. FORMAL TECHNIQUES FOR NETWORKED AND DISTRIBUTED SYSTEMS (FORTE XIV, Kluwer Academic Publishers (2001) 184–200
24. Pugliese, R., Tiezzi, F., Yoshida, N.: A symbolic semantics for a calculus for service-oriented computing. Electron. Notes Theor. Comput. Sci. **241** (July 2009) 135–164
25. Groote, J.F., Mathijssen, A., Reniers, M., Usenko, Y., Weerdenburg, M.V.: The formal specification language mcrl2. In: In Proceedings of the Dagstuhl Seminar, MIT Press (2007)