ARC — A Tool for Efficient Refinement and Equivalence Checking for CSP

Atanas N. Parashkevov and Jay Yantchev

ata,jty@cs.adelaide.edu.au
Department of Computer Science, University of Adelaide, SA 5005, Australia

Abstract— **This paper presents the design and implementation of ARC — a tool for automated verification of concurrent systems. The tool is based on the untimed CSP language, its semantic models and theory of refinement. We alleviate the combinatorial explosion problem using Ordered Binary Decision Diagrams (OBDDs) for the internal representation of complex data structures — sets and labeled transition systems (LTS). The semantically complex external choice operator is translated into the corresponding LTS using an optimized algorithm. This and some other implementation improvements allow verifying systems with up to 10^{33} states, which is consistent with the capabilities of other OBDD-based approaches.**

Compared to two existing CSP tools, FDR and MRC, ARC has fewer language restrictions and is more memory efficient. A performance comparison based on the n-schedulers and dining philosophers problems suggests that the checking algorithm of ARC is, in most cases, faster than those of the other tools.

Keywords— **Concurrent systems, verification, process algebras, CSP, OBDD**

I. Introduction

THE need for software tools supporting formal methods for concurrency has been widely acknowledged by the research community. Such tools are to perform automated transformations, proofs, refinement and verification in an integrated environment, much like modern CAD systems. A major problem with concurrent systems specification and verification is that the number of states in such a system may grow exponentially with the number of its concurrent components. This severely limits the size of the systems that can be handled by such tools.

The aim of this paper is to propose a concise representation of CSP process semantics which alleviates the combinatorial explosion problem and allows handling larger concurrent systems. *Ordered Binary Decision Diagrams* (or OBDDs) [2] are used to represent the LTS semantics of CSP processes. We introduce an efficient translation of the semantically complex external choice operator into an OBDD which is smaller, both in terms of size and number of OBDD variables, than previously used encodings [1]. Other performance enhancements, such as storing process refusals in an implicitly disjuncted form, result in lower memory and time requirements for the verification of complex CSP processes.

The techniques presented in this paper have been incorporated in a verification tool tentatively named ARC (Adelaide Refinement Checker). Compared to two existing CSP tools, FDR [10] and MRC [1], it imposes fewer language restrictions. A performance comparison based on well known "benchmark" examples shows that ARC in most cases is faster than the tools used in the comparison. The largest example ARC has been used to verify has about 10^{33} states which is consistent with the capabilities of other OBDD based verification tools [5], [7], [14] and far beyond the capacity of tools relying on explicit state enumeration [6], [10], [20].

II. Background

A. The CSP framework

Communicating Sequential Processes (CSP) has been proposed by Tony Hoare [11] and later developed by him and other researchers at Oxford University. The CSP framework consists of a well developed algebraic language [12], standard text-based representation [18], a hierarchy of semantic models [12], and a formal theory of refinement, equivalence and compositional verification [16]. This paper considers only a sufficiently powerful subset of CSP processes, defined as follows:

$$\begin{aligned} P \;=\; & STOP \mid SKIP \mid a \rightarrow P \mid P \,\Box\, Q \mid P \sqcap Q \mid P \,;\, Q \mid \\ & P \setminus A \mid ch\,?\,x \mid ch\,!\,x \mid P \underset{A}{\parallel} Q \mid P \,|||\, Q \mid \\ & ProcessName \mid f(P) \mid f^{-1}(P) \end{aligned}$$

A CSP program is considered to be a collection of (possibly recursive) process definitions in the form $ProcessName_i \stackrel{\text{def}}{=} ProcessTerm_i$.

The three semantic models of CSP processes form a hierarchy with an increasing degree of expressiveness and detail. In the basic *traces model* a process is considered to be just a set of traces. This captures all safety properties of a process, but cannot distinguish between deterministic and nondeterministic processes nor capture liveness properties such as deadlock. The *failures model* resolves this by extending the traces model with *refusal sets* — sets of events the process may refuse to engage in after a particular trace, thus giving a formal treatment of nondeterminism and deadlock in a very elegant way. Finally, the *failures-divergences model* captures also the possibility of a process entering an infinite unbroken loop of internal events — a phenomenon known as *divergence*.

CSP also formalizes the notion of *implementation* or *refinement* between processes. It is said that process Q *refines* process P (denoted $P \sqsubseteq Q$) if and only if all possible behaviors of Q are also possible behaviors of P. Since there are three models of CSP process behavior, there are also three increasingly stronger *refinement relations*: trace refinement, denoted $P \sqsubseteq_T Q$, failures refinement, denoted $P \sqsubseteq_F Q$, and failures-divergences refinement, denoted $P \sqsubseteq_{FD} Q$.

A specification in CSP is a process itself, which differs from most other verification approaches using some form of temporal logic for this purpose. This however is no restriction since a CSP process can be used not only for formalizing a particular concurrent system design but also as a description of a desired property. For every property or specification that can be formulated in any of the CSP models, there exists a most nondeterministic process $SPEC$ that captures that property. Then proving that a process $IMPL$ satisfies that property is reduced to demonstrating that $SPEC \sqsubseteq IMPL$. Traces refinement is the weakest relation of all but sufficient for verifying safety properties. Failures-divergences refinement is the strongest relation,

and if $SPEC \sqsubseteq_{FD} IMPL$, it is said that the $IMPL$ process is a *valid implementation* of the $SPEC$ process.

Similarly, CSP defines a stronger *equivalence relation* between processes. It is said that process P is equivalent to process Q (denoted $P = Q$) if and only if both $P \sqsubseteq Q$ and $Q \sqsubseteq P$ hold. Again, the equivalence relation can be applied in any of the three behavioral models in CSP.

B. Labeled transition systems

Since the refinement relations are defined in terms of the CSP models, an automated refinement/equivalence checking tool must first derive the formal semantics of the $SPEC$ and $IMPL$ processes from their algebraic descriptions. For various technical reasons it is most suitable to represent process semantics as a *labeled transition system* (LTS) [16].

A nondeterministic LTS L is a tuple $L = \langle S, A, R, I \rangle$, where S is the set of states in the LTS; A is the set of actions (events) in the LTS; $R \subseteq S \times A \times S$ is the set of labeled transition relations in the LTS; $I \subseteq S$ is the set of initial states of the LTS.

It is important to differentiate between a state in an LTS L and a state of the process P from which L has been derived. A state in L is simply any $\sigma \in S$. A state of P is the set of LTS states $\Sigma \subseteq S$ that are reachable from I via a particular sequence of visible events (transitions). The number of LTS states in Σ is a measure of the degree of nondeterminism contained in P. If P is a deterministic process then each state of P corresponds to a state in L.

For any transition relation R it is possible to define its transitive closure R^T:

$$R^T = \{\sigma_1 \overset{a_1 a_2 \ldots a_n}{\rightarrow} \sigma_{n+1} \mid \exists \{\sigma_i\}_{i=2}^{n} : \forall\, i \in \{1, 2, \ldots, n\}\ \sigma_i \overset{a_i}{\rightarrow} \sigma_{i+1} \in R\}$$

If a^* denotes a possibly empty sequence of a events and $\Sigma \subseteq S$, we define the following functions:

$$\mathsf{next}(\Sigma, L) = \{\langle a, \xi \rangle \mid \exists\, \sigma \in \Sigma : \sigma \overset{\tau^* a}{\rightarrow} \xi \in R^T\}$$

$$\mathsf{next\_events}(\Sigma, L) = \{a \mid \exists\, \sigma \in \Sigma, \theta \in S : \sigma \overset{\tau^* a}{\rightarrow} \theta \in R^T\}$$

$$\mathsf{next\_states}(\Sigma, C, L) = \{\theta \mid \exists\, a \in C, \sigma \in \Sigma : \sigma \overset{\tau^* a}{\rightarrow} \theta \in R^T\}$$

$$\mathsf{tau\_expand}(\Sigma, L) = \{\xi \mid \exists\, \sigma \in \Sigma : \sigma \overset{\tau^*}{\rightarrow} \xi \in R^T\}$$

$$\mathsf{tau\_expand\_reverse}(\Sigma, L) = \{\xi \mid \exists\, \sigma \in \Sigma : \xi \overset{\tau^*}{\rightarrow} \sigma \in R^T\}$$

The above functions are used extensively in ARC and are further discussed in Sections IV and V.

C. The state explosion problem

Explicit representations of LTS tend to grow in size and generation time exponentially with the number and complexity of concurrent processes. Consider n processes with m states each running concurrently with no synchronization between them. Such a system would have m^n distinct states. This is where the state explosion problem arises.

Several approaches have been suggested to alleviate this problem, such as *on-the-fly verification* [9], *stubborn set method* [19], *state compression* [16], [17], and others. The stubborn set and state compression methods are both based around the idea of state space reduction during a compositional build-up of the LTS [20], [17]. Another approach is to use OBDDs for *symbolic computations* over the state space and the transition relation of LTS. This method has made possible handling LTSs with 10^{20} and more states [5], [7].

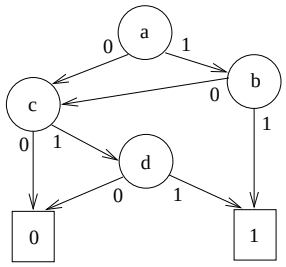

Fig. 1. An example OBDD

It should be noted that the OBDD approach is orthogonal to most other methods proposed in the literature. Therefore, it should be possible to combine the benefits of the symbolic approach with some of the state reduction methods, which may allow handling even larger state spaces than any of the discussed approaches alone.

D. Ordered Binary Decision Diagrams

An OBDD is a directed acyclic graph used to represent a boolean function $f : \mathcal{B}^n \longmapsto \mathcal{B}$, where $\mathcal{B}$ is the boolean domain: $\mathcal{B} = \{0, 1\}$ [2]. They provide canonical representation for boolean formulas and functions that is often much more compact than any of the normal forms. Two important properties of the OBDD are the strict total order of variable occurrence in the graph and that there are no identical subgraphs in the OBDD. An example OBDD is given on Figure 1, representing the boolean function $f(a, b, c, d) = (a \wedge b) \vee (c \wedge d)$ with the variable ordering $a \prec b \prec c \prec d$.

It has been shown that the size of the OBDD is very sensitive to the variable ordering [2], [3]. In the worst case it may be exponential to the number of its variables. Fortunately, some simple heuristics providing very good results have been found for several classes of boolean functions including representations of transition relations and n-bit adders [3].

The complete set of boolean, variable substitution and quantification operators can be implemented on OBDDs in a very efficient way [2]. With a suitable encoding scheme OBDDs also provide compact representation of other fundamental mathematical objects – sets, tuples, relations, transition systems, etc.

E. Overview of existing verification tools

Recently, a number of verification tools have been developed — SMV [14], Concurrency Workbench [6], FDR [10], ARA [20], MRC [1], the CCS tool [8], and others. Because of our choice of CSP as a specification language, FDR and MRC (Milano Refinement Checker) are most relevant to this work. Both of these perform only automatic refinement checking between two untimed CSP processes.

FDR, developed by Formal Systems (Europe) Ltd, is a commercially available software tool with X windows front-end and extensive debugging information. It has been successfully applied to a number of practical problems. The main drawback of its current version (FDR 1.42) is that the combinatorial explosion problem is not addressed and the size of the process state space is rather limited. Also, CSP processes in FDR are divided into low- and high-level categories and expressions like $(P \setminus A) \,\Box\, (Q \setminus B)$ are not permitted.

A paper by Roscoe et al. [17] describes some improvements planned for a new release of FDR (FDR 2.0), which is expected sometime in 1996. It will feature a new sophisticated, semi-automatic state compression algorithm constructing an implicit LTS representation called *generalized* LTS (GLTS). This should

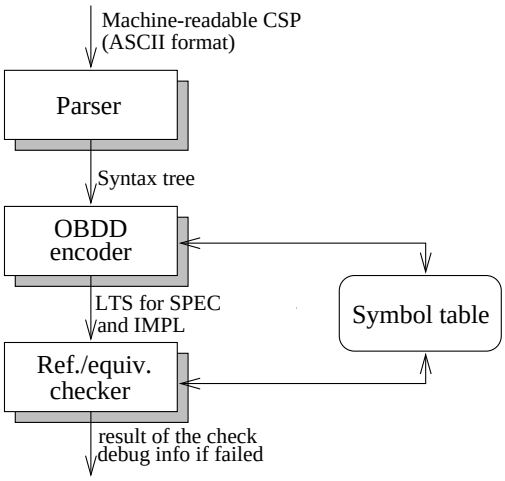

Fig. 2. The structure of ARC

allow the verification of systems with state spaces comparable to those handled by OBDD-based approaches.

The MRC tool has been developed at the University of Milano [1]. Designed as an alternative to an early prototype of FDR built at Inmos, it covers only a limited subset of the CSP language. It does not include mutual recursion, sequential composition, channels and process parameters. The MRC tool addresses the combinatorial explosion problem by using OBDDs as a representation of CSP process semantics. However, some deficiencies in its OBDD encoding and refinement checking algorithms result in severe performance bottlenecks and prevent it from exploiting the full potential of the OBDD approach.

III. The structure and implementation of ARC

The structure of our tool is shown on Figure 2. It contains three modules: a parser, an OBDD encoder and a refinement/equivalence checking module. A public domain CSP parser is used [18], which makes our input CSP format compatible with that of FDR. The original parser has been slightly modified in order to make the internal representation of the parse tree more appropriate to our needs.

The OBDD encoding module is responsible for the semantic check of the syntax tree and the generation of the LTS semantics for the specification and the implementation processes in an OBDD form. A detailed discussion of the functionality of this module is given in Section IV.

The refinement/equivalence checking module performs an exhaustive search on the product of the state spaces of the two LTS in order to prove or refute the relation being tested. This relation can be either a refinement or an equivalence in any of the three CSP models. This module is presented in detail in Section V.

A prototype implementation of ARC has been written in C using a public domain OBDD library developed at CMU. We are currently replacing the command line interface to the tool with a graphical user interface (GUI) based on Java/Motif.

IV. The OBDD encoder

A. OBDD variable ordering

Since the most complex element in the LTS tuple, R, is a set of transitions $\sigma \xrightarrow{a} \sigma'$, three OBDD variable vectors $\mathcal{S}$, $\mathcal{A}$, and $\mathcal{R}$ are required for encoding σ, a, and σ', respectively. Then S and I are encoded using the variable vector $\mathcal{S}$, and A is encoded using variables from $\mathcal{A}$. ARC also makes use of a fourth OBDD variable vector $\mathcal{F}$ for computing the failures of a process.

The variable ordering that has been chosen for ARC is:

$$s_0 \prec r_0 \prec s_1 \prec r_1 \prec \ldots \prec s_{k-1} \prec r_{k-1} \prec a_0 \prec a_1 \prec \ldots \prec a_{l-1} \prec f_0 \prec f_1 \prec \ldots \prec f_{m-1}$$

Various researchers have reported that interleaving of s_i and r_i variables gives best results for encoding transition relations [3]. Also, our experience with ARC shows that putting $\mathcal{A}$ at the end of $\mathcal{S}$ and $\mathcal{R}$ consistently leads to smaller OBDD sizes and faster verification as compared to putting $\mathcal{A}$ in front of $\mathcal{S}$ and $\mathcal{R}$. Note that this differs from the ordering proposed in [8], which is probably due to different encoding and verification algorithms. The exact ordering of the variables in $\mathcal{F}$ is not critical because implicit disjunction is used for storing process refusals (see Section V.B).

B. Deriving the LTS semantics

Derivation of the LTS semantics of a CSP process from its algebraic description is the main feature of the OBDD encoding module. A mapping function $\psi : \mathit{Process} \rightarrow \mathit{LTS}$ is constructed, where *Process* is the domain of valid CSP process expressions, and *LTS* is the domain of labeled transition systems. The mapping function is syntax-driven — for each CSP operator $\odot$ there is a corresponding operator $\odot_{lts}$ over the *LTS* domain, such that $\psi[\![P \odot Q]\!] = \psi[\![P]\!] \odot_{lts} \psi[\![Q]\!]$. In defining ψ, we closely follow the operational semantics of CSP [13]. For the rest of this section it is assumed that $\psi(P) = \langle S_P, A_P, R_P, I_P \rangle$, $\psi(Q) = \langle S_Q, A_Q, R_Q, I_Q \rangle$, $S_P \sqcap S_Q = \{\}$, and σ, σ_1 and σ_2 are distinct states.

We begin with the definitions of ψ for the primitive processes *STOP* and *SKIP*, and the prefix and renaming operators. They are all fairly intuitive:

$$\begin{array}{lcl}
\psi[\![STOP]\!] & = & \langle \{\sigma\}, \{\}, \{\}, \{\sigma\} \rangle \\
\psi[\![SKIP]\!] & = & \langle \{\sigma_1, \sigma_2\}, \{\surd\}, \{\sigma_1 \xrightarrow{\surd} \sigma_2\}, \{\sigma_1\} \rangle \\
\psi[\![a \rightarrow P]\!] & = & \langle \{\sigma\} \cup S_P, \{a\} \cup A_P, \\
 & & \{\sigma \xrightarrow{a} \sigma' \mid \sigma' \in I_P\} \cup R_P, \{\sigma\} \rangle, \sigma \notin S_P \\
\psi[\![f(P)]\!] & = & \langle S_P, \{f(a) \mid a \in A_P\}, \\
 & & \{\sigma \xrightarrow{f(a)} \sigma' \mid \sigma \xrightarrow{a} \sigma' \in R_P\}, I_P \rangle
\end{array}$$

The channel input and output operators are dealt with as in FDR [10]. If `T` is a discrete type of size n, then for each channel of type `T`, n prefix events are created — one for each value that may be sent over that channel. The LTS upon the channel input operator "branches" for each of these values using the external choice operator.

The definition of the internal choice operator joins all the elements in the LTS tuples:

$$\psi[\![P \sqcap Q]\!] = \langle S_P \cup S_Q, A_P \cup A_Q, R_P \cup R_Q, I_P \cup I_Q \rangle$$

The definition of the hiding operator requires a change in the labeled transition relation of the resulting LTS so that all the actions in the hiding set C are renamed to the internal τ event:

$$\begin{array}{lcl}
\psi[\![P \setminus C]\!] & = & \langle S_P, \\
 & & A_P \cup \{\tau\} - C, \\
 & & \{\sigma \xrightarrow{a} \sigma' \mid \sigma \xrightarrow{a} \sigma' \in R_P \wedge a \notin C\} \cup \\
 & & \{\sigma \xrightarrow{\tau} \sigma' \mid \sigma \xrightarrow{a} \sigma' \in R_P \wedge a \in C\}, \\
 & & I_P \rangle
\end{array}$$

The internal choice operator can be thought of as introducing *explicit* nondeterminism into the LTS representation by increasing the number of initial states, while the hiding operator leads

to *implicit* nondeterminism by introducing internal events. Distinguishing between these two is important for the correct definition and the efficient implementation of the external choice operator. Neither MRC nor FDR can handle this properly. MRC fails to acknowledge that external choice cannot be resolved by any number of internal events (cf. [1]), while FDR defines hiding as a high level operator and thus forbids CSP expressions like $(P \setminus A) \,\Box\, (Q \setminus B)$. A correct definition of ψ for the external choice operator fixes the shortcoming of MRC by adding two more terms (marked with $(*)$) to the expression:

$$\begin{aligned}
\psi[\![P \,\Box\, Q]\!] = \langle & \{(S_P \cup \{\xi_P\}) \times (S_Q \cup \{\xi_Q\}), \\
& A_P \cup A_Q, \\
& \{(\sigma_1,\sigma_2) \xrightarrow{\tau} (\sigma'_1,\sigma_2) \mid \sigma_2 \in S_Q \wedge && (*)\\
& \quad \sigma_1 \xrightarrow{\tau} \sigma'_1 \in R_P\} \cup && (*)\\
& \{(\sigma_1,\sigma_2) \xrightarrow{\tau} (\sigma_1,\sigma'_2) \mid \sigma_1 \in S_P \wedge && (*)\\
& \quad \sigma_2 \xrightarrow{\tau} \sigma'_2 \in R_Q\} \cup && (*)\\
& \{(\sigma_1,\sigma_2) \xrightarrow{a} (\sigma'_1,\xi_Q) \mid a \neq \tau \wedge \sigma_2 \in S_Q \wedge && (\dagger)\\
& \quad \sigma_1 \xrightarrow{a} \sigma'_1 \in R_P\} \cup && (\dagger)\\
& \{(\sigma_1,\sigma_2) \xrightarrow{a} (\xi_P,\sigma'_2) \mid a \neq \tau \wedge \sigma_1 \in S_P \wedge && (\dagger)\\
& \quad \sigma_2 \xrightarrow{a} \sigma'_2 \in R_Q\} \cup && (\dagger)\\
& \{(\sigma_1,\xi_Q) \xrightarrow{a} (\sigma'_1,\xi_Q) \mid \sigma_1 \xrightarrow{a} \sigma'_1 \in R_P\} \cup && (\ddagger)\\
& \{(\xi_P,\sigma_2) \xrightarrow{a} (\xi_P,\sigma'_2) \mid \sigma_2 \xrightarrow{a} \sigma'_2 \in R_Q\}, && (\ddagger)\\
& I_P \times I_Q\rangle
\end{aligned}$$

The above expression obviously requires some explanation. There are two new states $\xi_P, \xi_Q \notin S_P \cup S_Q$ added to the state spaces of, respectively, processes P and Q. The terms marked with $(*)$ capture the property of the general choice operator that it cannot be resolved by a sequence of internal events (i.e. neither P nor Q will be chosen after τ^*). The terms marked with $(\dagger)$ stand for the choice between P and Q; when this happens the process which has not been chosen makes a transition to the respective ξ state. Finally, the terms marked with $(\ddagger)$ represent the behavior of $P \,\Box\, Q$ after one of the processes has been already chosen.

This is still quite inefficient in that it requires building the product of the state spaces of P and Q. Even though this operation can be implemented relatively efficiently using OBDDs it will certainly increase the number of OBDD variables and OBDD nodes in the final LTS. The external choice operator is used quite frequently in CSP and therefore this inefficiency may become a serious bottleneck for larger examples.

To overcome this problem a new algorithmic approach has been developed. This approach only requires building the sum of the original state spaces while adding a small number of additional states. The algorithm works in two steps.

The first step transforms P and Q into P' and Q' by removing the implicit nondeterminism while preserving process semantics in the failures model. Thus, P' and Q' do not have internal actions possible from any of their initial states (although they may have internal actions possible afterwards). This is done by an LTS transformation function $\mathsf{tau\_remove} : LTS \rightarrow LTS$, which works as follows:

1. It first derives the set of states reachable from I by τ^* that have at least one possible visible transition or no transition at all (a $STOP$ state):

$$\begin{aligned}
\Sigma \;=\; & \{\sigma \mid \sigma \in \mathsf{tau_expand}(I, L) \wedge \\
& (\exists\, a \neq \tau : \sigma \xrightarrow{a} \xi \in R \vee \nexists\, a, \xi : \sigma \xrightarrow{a} \xi \in R)\}
\end{aligned}$$

2. If $\Sigma = \{\sigma_i\}_{i=1}^n$, then a new set of states $\Theta = \{\theta_i\}_{i=1}^n$ is created, such that $\Theta \cap S = \{\}$, and there is a one-to-one mapping from Θ to Σ: f_{map}

3. The new LTS L' is then constructed as follows:

$$\begin{aligned}
L' = \langle & S \cup \Theta, A, \\
& R \cup \{\theta \xrightarrow{a} \sigma \mid \theta \in \Theta \wedge \langle a, \sigma\rangle \in \mathsf{next}(\{f_{map}(\theta)\}, L)\} \\
& \Theta\rangle
\end{aligned}$$

Theorem 1: The transformation tau_remove preserves the failures semantics of CSP.

Proof: To prove this theorem, we have to introduce the functions tr_L, ref_L and $fail_L$ that compute, respectively, the traces, refusals and failures of a CSP process with LTS semantics $L = \langle S, A, R, I\rangle$ in a process state Δ. The equation for traces requires little explanation:

$$tr_L(\Delta, L) = \bigcup_{\delta \in \Delta} tr_L(\{\delta\}, L) =$$

$$\bigcup_{\delta \in \Delta} \bigcup_{a \in \mathsf{next\_events}(\{\delta\}, L)} \langle a^\frown tr(\mathsf{next\_states}(\{\delta\}, \{a\}, L), L)\rangle \quad (1)$$

Given the explicit nondeterminism contained in the set of states Δ, we have:

$$ref_L(\Delta, L) = \bigcup_{\delta \in \Delta} ref_L(\{\delta\}, L) \quad (2)$$

A node δ in an LTS L can have any number of visible and/or internal transitions possible from it in L. It models the initial state of the LTS semantics of the CSP process:

$$P = (\Box_{x \in C \cup D}\, x \rightarrow P(x)) \setminus D,\; C \cap D = \{\}$$

Using the algebraic laws of CSP [12], P can be rewritten as:

$$\begin{aligned}
P = ((\Box_{x \in C}\, x \rightarrow P(x)) \setminus D \,\Box\, (\Box_{x \in D}\, P(x)) \setminus D) \sqcap \\
(\sqcap_{x \in D}\, P(x) \setminus D) \quad (3)
\end{aligned}$$

For the simplicity of presentation, let us assume that for $x \in D$: $P(x)$ are deterministic in their initial state. Then, applying the laws of refusals in CSP [12], we derive:

$$\begin{aligned}
\forall\, x \in D : ref(P(x)) &= \bigcup_{t \in tr(P(x))} \{t_0\} \\
ref(\Box_{x \in D}\, P(x) \setminus D) &= \bigcup_{t \in tr(P(x)), x \in D} \{t_0\} \quad (4)
\end{aligned}$$

Applying the laws of refusals on equation 3 and using equation 4, we derive:

$$ref(P) = \{\{\Sigma - C - \bigcup_{t \in tr(P(x)), x \in D} \{t_0\}\}\} \cup \bigcup_{x \in D} ref(P(x)) \quad (5)$$

where Σ is the universal alphabet.

Let $\Theta = \{\theta \mid \delta \xrightarrow{\tau} \theta \in R\}$ is the set of states reachable from δ via a single internal transition. The equation 5 can now be formulated in the LTS semantics:

$$ref_L(\{\delta\}, L) = \{\{A - \mathsf{next\_events}(\{\delta\}, L)\}\} \cup \bigcup_{\theta \in \Theta} ref_L(\{\theta\}, L) \quad (6)$$

The above expression can be unfolded until all LTS states reachable from δ via a sequence of internal transitions are covered:

$$ref_L(\{\delta\}, L) = \bigcup_{\theta \in \mathsf{tau\_expand}(\{\delta\}, L)} \{\{A - \mathsf{next\_events}(\{\theta\}, L)\}\} \quad (7)$$

Equations 2 and 7 can be combined in one:

$$ref_L(\Delta, L) = \bigcup_{\theta \in \mathsf{tau\_expand}(\Delta, L)} \{\{A - \mathsf{next\_events}(\{\theta\}, L)\}\} \quad (8)$$

We are now prepared to formulate the equation for failures in the LTS semantics, using equations 1 and 8:

$$fail_L(\Delta, L) = \{(t, r) \mid t \in tr_L(\Delta, L) \wedge r \in ref_L(\Pi(\Delta, t, L), L)\} \quad (9)$$

where $\Pi(\Delta, t, L) = \{\pi_n \mid \exists \{\pi_i\}_{i=0}^{n=\sharp t} : \pi_0 \in \Delta \wedge \pi_i \stackrel{\tau^* t_i}{\rightarrow} \pi_{i+1} \in R^T\}$ is the set of LTS states reachable from Δ via a sequence of visible transitions t.

From the definition of tau_remove we conclude that:

$$\mathsf{next\_events}(I, L) = \mathsf{next\_events}(I', L') \quad (10)$$

$$\mathsf{next\_states}(I, \{a\}, L) = \mathsf{next\_states}(I', \{a\}, L') \quad (11)$$

Another property of L' that follows from the definition of tau_remove is that the states I' are unreachable after the first visible transition, therefore $R' \triangleright \{S' - I'\} = R$. From this and equations 10 and 11 we conclude that:

$$tr_L(I, L) = tr_L(I', L') \quad (12)$$

$$\forall\, t \in tr_L(I, L), t \neq \langle\rangle : \Pi(I, t, L) = \Pi(I', t, L') \quad (13)$$

From equations 9, 12, and 13 we conclude that:

$$fail_L(I, L) = fail_L(I', L') \Leftrightarrow ref_L(I, L) = ref_L(I', L') \quad (14)$$

Applying equation 8 to L and L', we can see that the difference between the resulting expressions for $ref_L(I, L)$ and $ref_L(I', L')$ is that the latter excludes the LTS nodes which only have internal transitions possible from them. Let δ is one such node; then, if $\Theta = \{\theta \mid \delta \stackrel{\tau}{\rightarrow} \theta \in R\}$, we have:

$$\mathsf{next\_events}(\{\delta\}, L) = \bigcup_{\theta \in \Theta} \mathsf{next\_events}(\{\theta\}, L) \Rightarrow$$

$$\forall\, \theta \in \Theta : \{A - \mathsf{next\_events}(\{\delta\}, L)\} \subseteq ref_L(\{\theta\}, L),$$

and since one of the basic refusals laws is:

$$X \in ref(P) \Rightarrow \forall\, Y \subseteq X : Y \in ref(P),$$

from equation 6 we derive:

$$\nexists\, a \neq \tau, \xi \in S : \delta \stackrel{a}{\rightarrow} \xi \in R \Rightarrow ref_L(\{\delta\}, L) = \bigcup_{\theta \in \Theta} ref_L(\{\theta\}, L) \quad (15)$$

From equations 8, 14 and 15 we derive that $fail_L(I, L) = fail_L(I', L')$, which proves the theorem. ∎

It is easy to see that L' has only visible transitions from any of its initial states. The cost of achieving this is the addition of n new states from the set Θ. In practice, however, some of the states in I and Σ may become unreachable from Θ and therefore can be safely discarded from S'. Thus, the number of additional states actually required by tau_remove is usually much less than n.

The second step in our algorithm combines P' and Q' using:

$$\begin{aligned}
\psi[\![P \,\Box\, Q]\!] &= \psi[\![P' \,\Box\, Q']\!] = \\
&= \langle S_{P'} \cup S_{Q'} \cup (I_{P'} \times I_{Q'}) - I_{P'} - I_{Q'}, \\
&\quad A_{P'} \cup A_{Q'}, \\
&\quad R_{P'} \cup R_{Q'} \cup \\
&\quad \{(\sigma^i_{P'}, \sigma^j_{Q'}) \stackrel{a}{\rightarrow} \theta \mid \sigma^i_{P'} \stackrel{a}{\rightarrow} \theta \in R_{P'} \vee \\
&\qquad \sigma^j_{Q'} \stackrel{a}{\rightarrow} \theta \in R_{Q'}\} \cup \\
&\quad \{(\sigma^1_{P'}, \sigma^1_{Q'}) \stackrel{\tau}{\rightarrow} (\sigma^1_{P'}, \sigma^1_{Q'}) \mid \qquad (*) \\
&\qquad \mathsf{divergent}(P) \vee \mathsf{divergent}(Q)\}, \qquad (*) \\
&\quad (I_{P'} \times I_{Q'})\rangle
\end{aligned}$$

where $I_{P'} = \{\sigma^i_{P'}\}_{i=1}^m$ are the initial states of P', and $I_{Q'} = \{\sigma^j_{Q'}\}_{j=1}^n$ are the initial states of Q'. The term marked with $(*)$ is used to preserve divergence in $\psi[\![P \,\Box\, Q]\!]$ whenever P or Q are divergent, thus fixing the side effect of tau_remove discussed above.

The product of the initial states of P' and Q' used above does not introduce a substantial overhead — it merely means that mn new LTS states are created. Since P' and Q' are derived using tau_remove, $I_{P'}$ and $I_{Q'}$ become unreachable in $\psi[\![P \,\Box\, Q]\!]$ and can be removed from the final product. Thus, the second step of our algorithm requires $mn - m - n$ new states. The total number of states required for the construction of $\psi[\![P \,\Box\, Q]\!]$ then becomes:

$$(|S_P| + m) + (|S_Q| + n) + (mn - m - n) = |S_P| + |S_Q| + mn,$$

which compares quite favorably to the cost of the initial construction $|S_P| \times |S_Q|$. In the extreme but not at all rare case when neither P nor Q have internal transitions possible from their initial states, at most one new state needs to be added.

The sequential composition operator requires all tick events to be hidden in P and the resulting process to behave like Q afterwards:

$$\begin{aligned}
\psi[\![P \,;\, Q]\!] = \langle & S_P \cup S_Q, \\
& A_P \cup A_Q \cup \{\tau\}, \\
& \{\sigma \stackrel{a}{\rightarrow} \xi \mid a \neq \surd \wedge \sigma \stackrel{a}{\rightarrow} \xi \in R_P\} \cup \\
& \{\sigma \stackrel{\tau}{\rightarrow} \xi \mid \sigma \in S_P \wedge \xi \in I_Q \wedge \\
& \quad \exists\, \theta \in S_P : \sigma \stackrel{\surd}{\rightarrow} \theta \in R_P\} \cup R_Q, \\
& I_P\rangle
\end{aligned}$$

The interleaving operator requires building the product of the state spaces of P and Q, and P and Q must only synchronize on the tick (termination) event:

$$\begin{aligned}
\psi[\![P \;|||\; Q]\!] = \langle & S_P \times S_Q, \\
& A_P \cup A_Q, \\
& \{(\sigma_1, \sigma_2) \stackrel{a}{\rightarrow} (\sigma'_1, \sigma_2) \mid a \neq \surd \wedge \sigma_2 \in S_Q \wedge \\
& \quad \sigma_1 \stackrel{a}{\rightarrow} \sigma'_1 \in R_P\} \cup \\
& \{(\sigma_1, \sigma_2) \stackrel{a}{\rightarrow} (\sigma_1, \sigma'_2) \mid a \neq \surd \wedge \sigma_1 \in S_P \wedge \\
& \quad \sigma_2 \stackrel{a}{\rightarrow} \sigma'_2 \in R_Q\} \cup \\
& \{(\sigma_1, \sigma_2) \stackrel{\surd}{\rightarrow} (\sigma'_1, \sigma'_2) \mid \sigma_1 \stackrel{\surd}{\rightarrow} \sigma'_1 \in R_P \wedge \\
& \quad \sigma_2 \stackrel{\surd}{\rightarrow} \sigma'_2 \in R_Q\}, \\
& I_P \times I_Q\rangle
\end{aligned}$$

The expression for the parallel composition operator is similar, but P and Q have to synchronize on the events in the set C:

$$\begin{aligned}
\psi[\![P \underset{C}{\parallel} Q]\!] = \langle & S_P \times S_Q, \\
& A_P \cup A_Q, \\
& \{(\sigma_1, \sigma_2) \stackrel{a}{\rightarrow} (\sigma'_1, \sigma_2) \mid \sigma_2 \in S_Q \wedge \\
& \quad \sigma_1 \stackrel{a}{\rightarrow} \sigma'_1 \in R_P \wedge a \notin C\} \cup \\
& \{(\sigma_1, \sigma_2) \stackrel{a}{\rightarrow} (\sigma_1, \sigma'_2) \mid \sigma_1 \in S_P \wedge \\
& \quad \sigma_2 \stackrel{a}{\rightarrow} \sigma'_2 \in R_Q \wedge a \notin C\} \cup \\
& \{(\sigma_1, \sigma_2) \stackrel{a}{\rightarrow} (\sigma'_1, \sigma'_2) \mid \sigma_1 \stackrel{a}{\rightarrow} \sigma'_1 \in R_P \wedge \\
& \quad \sigma_2 \stackrel{a}{\rightarrow} \sigma'_2 \in R_Q \wedge a \in C \cup \{\surd\}\}, \\
& I_P \times I_Q\rangle
\end{aligned}$$

Although both interleaving and parallel composition operators require building a product of two state spaces, the use of OBDDs allows the implementation of this with the standard set

operations. Furthermore, it can be proved that the OBDD representation of the resulting LTS grows, in the worst case, polynomially to the number of concurrent components [8]. This, however, does not hold for the intermediate OBDDs that are created during the refinement/equivalence checking. In our experience, it is the size of the intermediate OBDDs that limits the size of systems that can be verified using ARC.

C. State space management

A subtle but important issue of the OBDD encoding is state space management, that is, how the state vector $\mathcal{S}$ is being used during encoding and how new states are taken from the available state space when required. The size and complexity of the final OBDD representation of the LTSs can be very sensitive to effectiveness of the state space management scheme.

ARC uses an approach very similar to that in [8]. The state vector $\mathcal{S}$ is divided into a number of sub-vectors $\mathcal{S}_i$, one for each of the sequential processes P_i which constitute the CSP process P. Within the sub-vector $\mathcal{S}_i$ each distinct state of P_i is assigned its own unique boolean formula. The latter is obtained directly from the binary representation of a natural number. The length of each $\mathcal{S}_i$ is exactly $\lceil \log_2(|S_{P_i}|) \rceil$, i.e. the encoding of each sequential process uses the minimum number of OBDD variables possible. The sub-vectors $\mathcal{S}_i$ are gradually merged by the parallel composition operators in the definition of P until the final vector $\mathcal{S}$ is obtained. As it is demonstrated in Section VI, this simple and straightforward state space management scheme provides reasonable results.

V. The refinement/equivalence checking module

A. Computation of some basic functions

This subsection briefly describes the implementation of the LTS functions defined in Section II.B. If Σ and L are in OBDD representation, the next function can be computed as:

$$\mathsf{next}(\Sigma, L) = \mathcal{R}\,to\mathcal{S}(\exists\,\mathcal{S}(\Sigma \wedge R))$$

where $\mathcal{R}\,to\mathcal{S}$ is a variable substitution function which maps all variables from the $\mathcal{R}$ vector to the corresponding variables of the $\mathcal{S}$ vector, and $\exists$ and $\wedge$ are the standard OBDD functions. Computation of the next_events and next_states functions is done in a similar way. The above functions resemble the next-state relation widely used in symbolic model checking [5].

Computation of tau_expand and tau_expand_reverse is slightly more complicated. It involves finding the fixed point of the respective expressions:

$$\mathsf{tau\_expand}(\Sigma, L) = \mathrm{fixpt}(\Sigma = \Sigma \cup \{\xi \mid \exists\,\sigma \in \Sigma : \sigma \xrightarrow{\tau} \xi \in R\})$$
$$\mathsf{tau\_expand\_reverse}(\Sigma, L) = \mathrm{fixpt}(\Sigma = \Sigma \cup \{\xi \mid \exists\,\sigma \in \Sigma : \xi \xrightarrow{\tau} \sigma \in R\})$$

The existence of both fixed points can be easily proved based on the finite number of states in S. For improved performance, both tau_expand and tau_expand_reverse actually use a stripped down transition relation $R_\tau \subseteq R$:

$$R_\tau = \{\sigma_1 \xrightarrow{\tau} \sigma_2 \mid \sigma_1 \xrightarrow{\tau} \sigma_2 \in R\}$$

B. Computation of refusals

A refusal of a process in CSP is the set of all sets of events it can refuse after a particular trace. In ARC refusals are precomputed as a relation $\mathsf{ref} : S \times 2^{|A|} \rightarrow \mathcal{B}$. Its OBDD representation uses the state vector $\mathcal{S}$ for encoding states and $|A|$ OBDD variables from the vector $\mathcal{F}$. The refusals of any process state $\Sigma \subseteq S$ are then computed using a single OBDD operation — relational product:

$$\mathsf{refusals}(\Sigma, L) = \exists\,\mathcal{S}(\Sigma \wedge \mathsf{ref}(L)) \tag{16}$$

The derivation of ref involves obtaining the set of states $\Sigma_a \in S$ that can accept the event a, for each $a \in A$ using:

$$\begin{aligned}\Sigma_a(L) \quad = \quad & \{\sigma \mid \exists\,\xi \in S : \sigma \xrightarrow{a} \xi \in R\} \cup \\ & \{\sigma \mid \exists\,\xi \in S : \sigma \xrightarrow{\tau^+ a} \xi \in R^T\}\end{aligned}$$

The first part of the above expression is computed directly from the transition relation and the second one is derived from the first using the tau_expand_reverse function. The relation ref is then computed as:

$$\mathsf{ref}(L) = \bigvee_{a \in A} (\Sigma_a(L) \wedge f_a \vee \neg\Sigma_a(L)) \tag{17}$$

where f_a is the variable from $\mathcal{F}$ corresponding to the event a, and $\neg\Sigma_a(L)$ is used as a shortcut OBDD operation for $S - \Sigma_a(L)$.

The initial implementation of ARC has shown that the OBDD representation of ref requires a very large number of OBDD nodes. In fact, it grows prohibitively fast with the number of events in A and the complexity of the LTSs, and changing the variable ordering has little or no effect on the size of ref. However, the number of OBDD nodes required for the subterms of equation 17 is quite small. This observation has pointed out the solution to this problem. Instead of keeping ref as one composite relation it is stored as an implicit disjunction of the relations $\mathsf{ref}_a(L) = \Sigma_a(L) \wedge f_a \vee \neg\Sigma_a(L)$. Equation 16 is then converted to:

$$\mathsf{refusals}(\Sigma, L) = \exists\,\mathcal{S}(\Sigma \wedge (\bigvee_{a \in A} \mathsf{ref}_a(L))) = \bigvee_{a \in A} \exists\,\mathcal{S}(\Sigma \wedge \mathsf{ref}_a(L))$$

Keeping boolean functions as an implicit disjunction (or conjunction) of its components is not a new idea for reducing the size of an OBDD. This approach has been applied, for example, for reducing the size of a complex LTS [4]. In the case of ARC this method leads to considerable space savings while incurring negligible computation overhead.

C. Computation of divergences

The set of divergent states in an LTS is the set of all states which are either a part of a closed τ-loop or can reach a closed τ-loop via τ^*. ARC precomputes a relation $\mathsf{div} : S \rightarrow \mathcal{B}$ using a method described in an extended version of [1]. The check whether a process state $\Sigma \subseteq S$ is divergent is then quite straightforward:

$$\mathsf{divergent}(\Sigma, L) \Leftrightarrow \Sigma \cap \mathsf{div}(L) \neq \{\}$$

D. The refinement/equivalence checking algorithm

Roscoe [16] suggests the transformation of the specification process LTS into a so called *normal form LTS* before the actual refinement checking. The normal form is derived from the nondeterministic LTS by constructing the corresponding deterministic LTS with no hidden transitions and then joining states belonging to same equivalence classes. Each state of the CSP process corresponds to exactly one state of the normal form LTS, which considerably facilitates the refinement checking. As a drawback, the computation of the normal form is a potential source of combinatorial explosion.

The advantage of using a normal form LTS instead of a nondeterministic LTS is that this avoids working with sets of states

```
type Models: (Traces, Failures, Fail_Div);
type Relation: (Refinement, Equivalence);

boolean function
RefineCheck(P, Q: LTS, CheckType: Models, CheckRel: Relation) {
  type Event: A_P; -- must be the same as A_Q
  type EventList: list of Event;
  type pState: P S_P; -- P is the specification process
  type qState: P S_Q; -- Q is the implementation process

  var pending, checked: list of ⟨pState, qState, EventList⟩;
  var p, p1: pState;
  var q, q1: qState;
  var trace: EventList;
  var event: Event;

  pending := {⟨I_P, I_Q, ⟨⟩⟩}; -- We start from the pair of initial states
  checked := {}; -- Initially, we have checked nothing
  while (pending != {}) {
    -- Get the first pending pair of states
    ⟨p, q, trace⟩ := head(pending);
    pending := tail(pending);
    if (CheckRel = Refinement) then {
      if (next_events(q, Q) ⊈ next_events(p, P)) then
        -- Traces error after ⟨trace⟩
      elsif (CheckType > Traces and
             refusals(q, Q) ⊈ refusals(p, P)) then
        -- Failures error after ⟨trace⟩
    } else {
      if (next_events(q, Q) ≠ next_events(p, P)) then
        -- Traces error after ⟨trace⟩
      elsif (CheckType > Traces and
             refusals(q, Q) ≠ refusals(p, P)) then
        -- Failures error after ⟨trace⟩
    }
    if (CheckType = Fail_Div and divergent(q, Q) and
        not divergent(p, P)) then
      -- Divergence error after ⟨trace⟩
    checked := cons(checked, ⟨p, q, trace⟩);
    for (event in next_events(q, Q)) {
      p1 := next_states(p, event, P);
      q1 := next_states(q, event, Q);
      if (∄ X : ⟨p1, q1, X⟩ ∈ checked ∪ pending) then
        pending := cons(pending, ⟨p1, q1, cons(trace, event)⟩);
    }
  }
  if ("no errors were encountered") then
    return TRUE; -- Refinement/equivalence check succeeds
  else
    return FALSE; -- Refinement/equivalence check fails
}
```

Fig. 3. The refinement/equivalence checking algorithm

during the refinement checking as this may cause a serious performance bottleneck. However, OBDDs provide compact representation of sets of states, together with efficient algorithms for operations on such sets. This is why our refinement checking procedure, unlike that of FDR, works directly on the nondeterministic LTS derived from the specification and implementation processes as described in the previous section. In this sense our method resembles the *symbolic model checking* approach introduced in [5].

The general algorithm of our refinement checking procedure is shown on Figure 3. It resembles the one used in MRC with some minor modifications. The head, tail and cons are the standard list functions, and next_events and next_states are defined as in Section II.B.

The whole functionality of the verification algorithm is contained within the while loop which purpose is to go through all pairs of states in $S_P \times S_Q$ reachable from the initial pair (I_P, I_Q). Two lists of pairs are maintained, checked for the pairs that have been checked, and pending for the pairs that are still to be checked. Each run of the body of the loop performs one step in the verification on a pair of states. The conditions checked at each step depend on the model and the type of relation chosen for verification.

If the check for the current pair is successful, all pairs of states that are reachable via a single visible transition and have not been encountered yet are added to pending. If a refinement is not successful, the user is provided with a list of traces of increasing length after which the specification and implementation processes behave differently. In other words, the verification analysis goes as far as possible. We believe that this may be more helpful than reporting only the first error encountered (as in FDR). For example, if a refinement check reveals failures (liveness) errors but no traces (safety) ones (like in the n-philosophers problem), it provides some insights into the nature of the problem and the possible solutions.

Problem	States	MRC	CCS tool	ARC
12-sched	5.9×10^{10}	2488	1213	1219
14-sched	2.9×10^{12}	3242	1549	1560
16-sched	1.4×10^{14}	4096	1960	1944
18-sched	7.0×10^{15}	5038	2337	2356
20-sched	3.4×10^{17}	6080	2810	2827
30-sched	9.7×10^{25}	crash	—	5735
3-phil	2.7×10^{4}	1037	—	596
5-phil	2.4×10^{7}	3045	—	1652

Table 1: Number of OBDD nodes in the LTS

VI. Performance results

This section presents a performance evaluation of ARC compared to some existing process algebra verification tools — MRC, FDR and the CCS tool described in [8]. Two well known examples are used: the n-schedulers (n-sched) problem from [15] and the n dining philosophers (n-phil) problem from [12]. Both examples are suitable for performance comparison because the size of the verification problem can be easily adjusted by changing n.

The experiments with ARC, MRC and FDR have been done on a Sparcstation 10. The data for the CCS tool is taken from [8], where the execution times are given on a machine approximately equivalent to a SUN 3/60.

An important measure of the efficiency of the OBDD encoding algorithm is the number of OBDD nodes in the final LTS. A smaller OBDD means faster operations on the LTS it represents [2] and therefore faster refinement/equivalence checking. Table 1 summarizes the results for MRC, ARC, and the CCS tool (FDR does not use OBDDs). It shows that ARC produces an OBDD that is half the size of the OBDD generated by MRC and approximately the same size as that obtained by the CCS tool. This result is very encouraging given that the CSP definition of the n-schedulers problem (originally formulated in CCS) requires one extra process to insert a token into the scheduler ring in the beginning.

A performance comparison of the verification algorithms of ARC, MRC, FDR and the CCS tool based on the n-schedulers example is given on Figure 4. A number of observations can be made from the figure. Firstly, ARC is faster than any of the tools used in the comparison even though the tools are run on slightly different architectures with different performance characteristics. Secondly, the rate of growth of the time required for the refinement/equivalence checking as a function of n is

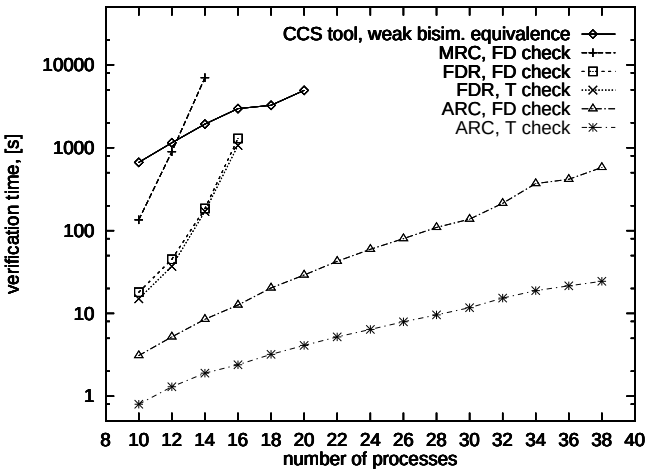

Figure 4: Verification times for the n-sched example

Tool	ARC		MRC	FDR	
Model	traces	fail.-div.	fail.-div.	traces	fail.-div.
3-phil	0.4	0.8	3.1	9	11
4-phil	2.3	5.2	28.3	11	12
5-phil	21.9	47.1	268	13	13

Table 2: Verification times for the n-phil example

lowest for ARC; that of the CCS tool is only marginally higher. Thirdly, FDR exhibits the typical behaviour of a tool using explicit state enumeration — its checking time grows from 18 seconds for the 10-sched example to more than 2 hours for the 18-sched example. Finally, the MRC tool clearly shows the worst verification times and growth rate of all tools used in the comparison.

A slightly different picture shows a performance comparison of the verification algorithms of ARC, MRC and FDR based on the n-phil example. The results of the experiments are given in Table 2. Most interesting is the 5-phil example, for which FDR outperforms ARC. There are three reasons for this. The first one is that FDR, unlike ARC, stops after the first error has been encountered. The second reason is the normalization procedure of FDR which reduces the specification to a single state LTS. The third reason is discussed in the next section.

VII. Discussion

An interesting and important question is: what determines the performance of the refinement/equivalence checking algorithm used in ARC? The answer to this question would suggest ways for further optimization of the algorithm.

The key to the answer is in the analysis of the algorithm on Figure 3. The refinement/equivalence checking time is proportional to the number of times the body of the while loop is executed, i.e. the number of pairs of states that are reachable from the pair of initial states. The number of steps in the check of the n-sched example is $2n+1$, whereas for the n-phil example it grows fast from 154 for 3 philosophers to 4474 steps for 5 philosophers. This explains why ARC's verification times for the n-phil problem grow so fast with n.

The only way to reduce the number of steps required for a refinement checking that we know of is state reduction [17], [19]. We plan to implement such algorithms in a future release of ARC.

Also, since the functions next_events, refusals and state_after work with the OBDD representing the LTS of the process, the time required for the checking is proportional to the size of that OBDD (this is mentioned in the previous section). Therefore, any optimization that reduces the number of nodes in that OBDD also speeds up the refinement/equivalence checking.

VIII. Conclusions

This paper presented an approach to the refinement checking of CSP processes that alleviates the combinatorial explosion problem by using OBDDs for the internal representation of complex data structres (sets and labeled transition relations). The ideas presented have been incorporated into the ARC tool and proven to lead to considerable time and space benefits for both the derivation of the CSP process semantics and the checking algorithm. Since the definition of the mapping function ψ is based on the operational semantics of CSP, a similar approach could be applied to any other process algebra with little modification.

References

[1] G. Barrett, M. N. Marcigaglia, G. Tateo, and M. Vaccari. A tool for checking refinement between finite-state CSP processes. In *Transputer Applications and Systems '94, Proceedings of the 1994 World Transputer Congress*. IOS Press, 1994.

[2] R. E. Bryant. Graph based algorithms for boolean function manipulation. *IEEE Trans. Computers*, C-35(8):677–691, August 1986.

[3] R. E. Bryant. Symbolic boolean manipulation with ordered binary decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.

[4] J. R. Burch, E. M. Clarke, and D. E. Long. Symbolic model checking with partitioned transition relations. In *VLSI '91*, pages 49–58, 1991.

[5] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98:142–170, 1992.

[6] R. Cleaveland, J. Parrow, and B. Steffen. The Concurrency Workbench: A semantics-based tool for the verification of concurrent systems. *ACM TOPLAS*, 15(1):36–72, January 1993.

[7] W. Damm, H. Hungar, P. Kelb, and R. Schlör. Statecharts: Using graphical specification languages and symbolic model checking in the verification of a production cell. In *Formal Development of Reactive Systems*, pages 131–149. Lecture Notes in Computer Science, 891, Springer Verlag, 1995.

[8] R. Enders, T. Filkorn, and D. Taubner. Generating BDDs for symbolic model checking in CCS. In *CAV '91*, pages 203–213. Lecture Notes in Computer Science, 575, Springer Verlag, 1991.

[9] J.-C. Fernandez and L. Mounier. A tool set for deciding behavioral equivalences. In *Concur '91*, pages 23–42. Lecture Notes in Computer Science, 527, Springer Verlag, 1991.

[10] Formal Systems (Europe) Ltd. *FDR: User Manual and Tutorial, version 1.3*, August 1993.

[11] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[12] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[13] He Jifeng and C. A. R. Hoare. From algebra to operational semantics. *Information Processing Letters*, 45:75–80, 1993.

[14] K. L. McMillan. *Symbolic Model Checking — an approach to the state explosion problem*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1992.

[15] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[16] A. W. Roscoe. Model-checking CSP. In *A Classical Mind, Essays in Honor of CAR Hoare*. Prentice Hall, 1994.

[17] A. W. Roscoe, P. H. B. Gardiner, M. H. Goldsmith, J. R. Hulance, D. M. Jackson, and J. B. Scattergood. Hierarchical compression for model-checking CSP or how to check 10^{20} dining philosophers for deadlock. In *TACAS'95 Workshop*, pages 133–152. Lecture Notes in Computer Science, 1019, Springer Verlag, 1995.

[18] B. Scattergood. *A Parser for CSP*. Oxford University, UK, December 1992.

[19] A. Valmari and M. Clegg. Reduced labeled transition systems save verification effort. In *Concur '91*, pages 526–540. Lecture Notes in Computer Science, 527, Springer Verlag, 1991.

[20] A. Valmari, J. Kemppainen, M. Clegg, and M. Levanto. Putting advanced reachability analysis techniques together: the "ARA" tool. In *FME'93*, pages 597–616. Lecture Notes in Computer Science, 670, Springer Verlag, 1993.