Electronic Communications of the EASST
Volume 23 (2009)

Proceedings of the Ninth International Workshop on Automated Verification of Critical Systems (AVOCS 2009)

Faster FDR Counterexample Generation Using SAT-Solving

H. Palikareva, J. Ouaknine and A. W. Roscoe

15 pages

Guest Editor: Markus Roggenbach
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Faster FDR Counterexample Generation Using SAT-Solving

H. Palikareva, J. Ouaknine and A. W. Roscoe

Oxford University Computing Laboratory, Oxford, UK

Abstract: With the flourishing development of efficient SAT-solvers, bounded model checking (BMC) has proven to be an extremely powerful symbolic model checking technique. In this paper, we address the problem of applying BMC to concurrent systems involving the interaction of multiple processes running in parallel. We adapt the BMC framework to the context of CSP and FDR yielding bounded refinement checking. Refinement checking reduces to checking for reverse containment of possible behaviours. Therefore, we exploit the SAT-solver to decide bounded language inclusion as opposed to bounded reachability of error states, as in most existing model checkers. We focus on the CSP traces model which is sufficient for verifying safety properties. We present a Boolean encoding of CSP processes resting on FDR's hybrid two-level approach for calculating the operational semantics using supercombinators. We describe our bounded refinement-checking algorithm which is based on watchdog transformations and incremental SAT-solving. We have implemented a tool, SymFDR, written in C++ which uses FDR as a shared library for manipulating CSP processes and the state-of-the-art SAT-solver MiniSAT. Experiments indicate that in some cases, especially for complex combinatorial problems, SymFDR significantly outperforms FDR.

Keywords: CSP, FDR, concurrency, process algebra, Bounded Model Checking, SAT-solving, safety properties

1 Introduction

Model checking techniques can be partitioned into those which are *symbolic*, based on abstract representation of sets of states, and those which are based on *explicit* examination of individual states. The former generally represent sets of states as formulae in Boolean logic and use techniques such as SAT-solving and BDD manipulation to decide checks. The latter can be enhanced by techniques such as hierarchical state-space compression and partial-order methods. The main obstacle when applying these approaches in practice is the *state-space explosion problem* by which the number of states in a system grows exponentially with the number of parallel components and also the number and bit sizes of data values.

FDR [Ros94, G+05] is a long-established tool for the refinement checking of CSP [Hoa85, Ros98]. When deciding whether a proposed implementation process *Impl* refines a normalised specification process *Spec*, FDR follows algorithms exploring the Cartesian product of the state spaces of *Spec* and *Impl* in a way comparable to conventional model checking. Therefore, until now, FDR has followed the explicit model checking approach. There has been, however, some work on the symbolic model checking of CSP [PY96, SLDS08].

This paper reports our attempts to integrate SAT-based bounded model checking [BCCZ99] into FDR. We show how the same internal structures used in FDR's two-level representation of state spaces can be translated readily into Boolean logic. Within the scope of this paper, we only consider the translation of *trace* refinement to SAT checking.

The result is a prototype tool SymFDR which, when combined with state-of-the-art SAT-solvers such as MiniSAT [ES03a, EB05], sometimes outperforms FDR by a significant margin when finding counterexamples. We compare the performance of SymFDR with the performance of FDR, FDR used in a non-standard way, PAT [SLD08] and, in some cases, NuSMV [CCG+02], Alloy Analyzer [Jac06] and straight SAT encodings of the problems under consideration.

The remainder of the paper is organised as follows. In Section 2, we set out the necessary background on CSP and FDR's two-level strategy for performing refinement checks. We briefly describe the ideas underlying BMC. In Section 3, we show how to adapt the watchdog approach [RGM+03] to BMC, while in Section 4, we summarise the methods we use to translate FDR's supercombinator representation of a state machine into input for a SAT-solver. Section 5 gives details of how SymFDR is built on top of this, and Section 6 offers experimental comparisons.

2 Preliminaries

2.1 CSP and FDR

In this section, we assume that the reader is familiar with CSP and we therefore give only a brief overview of CSP and FDR. The interested reader is referred to [Ros98]. Furthermore, we restrict our focus exclusively to the traces model, intentionally omitting information about other more expressive models of CSP.

2.1.1 CSP Syntax

In this section, we recall the core syntax of CSP. Let Σ be a finite alphabet of (visible) events with $\tau, \checkmark \notin \Sigma$. The internal action τ occurs silently and is invisible outside a process. $\checkmark$ denotes a successful termination of a process. In what follows, we assume that $a \in \Sigma$, $A \subseteq \Sigma$ and $B \subseteq \Sigma^{\checkmark} = \Sigma \cup \{\checkmark\}$. $R \subseteq \Sigma \times \Sigma$ denotes a renaming relation on Σ.

Definition 1 A CSP process is defined recursively via the following grammar:

$$\begin{aligned} P := \quad & STOP \mid SKIP \mid DIV \mid x : A \to P(x) \mid P_1 \Box P_2 \mid P_1 \sqcap P_2 \mid \\ & P_1 \underset{B}{||} P_2 \mid P_1 ; P_2 \mid P \setminus A \mid P[\![R]\!] \mid \mu P \bullet F(P) \end{aligned}$$

CSP_M converts core CSP into an ASCII form and adds several further operators and an extensive functional language. SymFDR supports the full CSP_M syntax, except that it cannot at present handle scripts using the function *chase*.

2.1.2 Denotational Semantics

CSP supports a hierarchy of several denotational semantic models. Each of them describes a process in terms of the observable behaviours it can exhibit. All denotational models are

compositional in the sense that the denotational value of each process can be computed in terms of the denotational values of its subcomponents.

In the traces model, a process P is identified with the set of its finite traces, denoted by $traces(P)$. Intuitively, a trace of a process is a sequence of visible actions that the process can perform. The set of traces of a process is non-empty and prefix-closed.

There are two different approaches for obtaining the set $traces(P)$ — either by constructing it inductively from the traces of its subcomponents, or by extracting it from the operational semantics. Refer to [Ros98] for the rules underlying the first approach. Since denotational values of processes are rather complex and often infinite, FDR calculates the behaviours of a process from its standard operational representation which is justified by semantic models being congruent to it. The congruence theorems are presented and proven in [Ros98].

2.1.3 Operational Semantics

The operational semantics models CSP processes as labelled transition systems (LTS's), with nodes denoting processes and labels denoting visible or τ actions. Since the LTS representation is not unique, in terms of the operational semantics, two processes are considered equivalent if they are strongly bisimilar [Ros98]. The operational semantics is calculated by repeatedly applying a set of inference rules, called *firing rules*. Firing rules provide recipes for constructing an LTS out of a CSP description of a process. The recipes define how processes can evolve by calculating the initial actions available at each node and the possible results after performing each action. The reader is referred to [Ros98] for more information.

Extracting Behaviours from Operational Semantics. We now present how behaviours, in our case – traces, can be retrieved from the operational semantics of a process.

Formally, a labelled transition system is a quadruple $M = \langle S, s_0, L, T \rangle$, where S is a finite set of states, $s_0 \in S$ is the initial state, L is a finite set of labels, $T \subseteq S \times L \times S$ is the transition relation.

For convenience, we write $s \xrightarrow{l} s'$ instead of $(s,l,s') \in T$. Furthermore, we write $s \xrightarrow{l}$ if there exists $s' \in S$, such that $s \xrightarrow{l} s'$. For $s \in S$ and $l \in L$, we define $Post(s,l) = \{s' \in S | s \xrightarrow{l} s'\}$ — the set of direct *l-successors* of s. M is then is deterministic if, for any $s \in S$ and $l \in L$, $|Post(s,l)| \leq 1$. An execution of M is a finite or an infinite alternating sequence of states and events $\pi = s_0 l_1 s_1 l_2 \ldots l_n s_n \ldots$ such that s_0 is the initial state and for all i, $s_i \xrightarrow{l_{i+1}} s_{i+1}$.

Let P be a finite-state process and $OS_P = \langle S^P, s_0^P, L^P = \Sigma^{\tau,\checkmark}, T^P \rangle$ be the LTS underlying the operational semantics of P. We denote by α_P the set of all visible events that P can perform, i.e. $\alpha_P = \Sigma^{\checkmark}$. We write $\Sigma^{*\checkmark}$ to denote the set of finite words over Σ which might end with $\checkmark$, and similarly, $(\Sigma^{\tau})^{*\checkmark}$. For $p,q \in S^P$, we use the following notation:

- $initials(p) = \{l \in \Sigma^{\checkmark} | p \xrightarrow{l}\}$, i.e. $initials(p)$ is the set of visible events that can be communicated from the state p.
- for $t = \langle x_i | 0 \leq i < n \rangle \in (\Sigma^{\tau})^{*\checkmark}$, we write $p \xmapsto{t} q$ if there exists a sequence of states $p_0, p_1, \ldots, p_n$, such that $p_0 = p$, $p_n = q$ and $p_k \xrightarrow{x_k} p_{k+1}$ for $k \in \{0, \ldots, n-1\}$.
- for $t \in \Sigma^{*\checkmark}$, we write $p \Longrightarrow q$ if there exists $t' \in (\Sigma^{\tau})^{*\checkmark}$, such that $p \xmapsto{t'} q$ and $t = t' \upharpoonright \Sigma^{\checkmark}$, i.e. t is t' with all the τ's removed.

Then, we define $traces(P) = \{t \in \Sigma^{*\checkmark} | \exists q \in S^P . s_0^P \Longrightarrow q\}$.

The Two-Level Approach. In fact, FDR exploits a hybrid high-/low- level approach for calculating the operational semantics of a process [Ros08]. Generally, the low level comprises all true recursions, the high level – processes composed by parallel composition, hiding and renaming, although the dividing line is a bit more complex. For each process compiled on the low level, an explicit LTS is produced, following the firing rules. Compiling on the high-level is called *supercompiling*. It is based on calculating a set of rules for turning a combination of LTS's into a single LTS, without explicitly constructing it. For most practical examples, the result of supercompilation is a high-level structure.

The high-level structure consists of two parts. The first one is a process tree with leaves – low-level compiled LTS's, and internal nodes – CSP operators such as hiding, renaming or parallel composition. Each node, even if internal, represents a process and is interpreted as an LTS with its behaviours deducible from the behaviours of its children on-the-fly. The second part of the high-level structure is a set of rules mapping actions of a number of leaf processes to an event-outcome of the composite root process [Ros98]. Those rules are called *supercombinators*. In what follows, we use the notions of supercombinators and rules interchangeably. Within a supercombinator, each process can participate with a visible event, a silent action τ, or not be involved at all. The supercompiler generates the following types of rules [Ros98, RRS+01]:

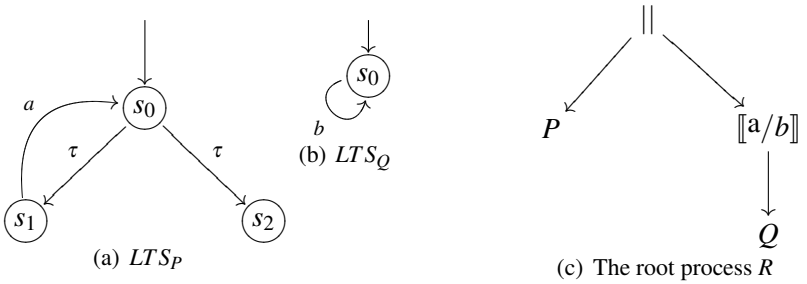
- a rule for a leaf willing to perform a τ which promotes a τ action of the root process
- rules using visible actions

Note that the visible actions that the leaf processes perform need not be the same if hiding or renaming is involved in the combination being modelled. For example, if $P = a \rightarrow P$ and $Q = b \rightarrow Q$, then if P performs a and Q performs b, $P \underset{\{a\}}{||} Q[\![a/b]\!]$ can perform a, where $Q[\![a/b]\!]$ is the process Q with the event b being renamed to a. Hence, (a,b,a) is a valid rule for the root process $P \underset{\{a\}}{||} Q[\![a/b]\!]$ with leaves P and Q: the first two elements of the rule triple represent the actions of P and Q, respectively, the last element provides the event-outcome of $P \underset{\{a\}}{||} Q[\![a/b]\!]$.

In FDR, at every particular step the leaf processes can be either *switched on* or *switched off*. Processes are switched on if they are currently under a CSP operator that makes their actions immediately relevant for the action of the overall system. Processes are switched off if they are under a CSP operator that does not need their actions to deduce the resulting action of the system. For instance, in $P_1|||P_2$, both P_1 and P_2 are switched on. In $P_1;P_2$, P_2 is switched off until P_1 performs $\checkmark$. In $a \rightarrow (P_1|||P_2)$, both P_1 and P_2 are initially switched off until a is communicated.

We refer to the different configurations of switched on and switched off leaf processes as *formats*. In the worst case, there could be exponentially many different formats, but in practice this is rarely the case. In FDR, the set of supercombinators is partitioned with respect to the existing formats. Hence, supercombinators can be viewed as dynamic or static, depending on whether they switch the system to another format after being triggered or not.

A state of the root high-level process, also called a *configuration*, is a tuple of the current states of its leaf processes. When running the root process, FDR computes its initial actions by checking which supercombinators are enabled from the current configuration and the current format of the root. A supercombinator might be disabled if not all leaf processes are able to communicate the event they are responsible for within the supercombinator. Hence, the operational semantics



(a) LTS_P (b) LTS_Q (c) The root process R

Figure 1: Example OS

of the root can be considered an implicit LTS, whose transitions can be switched on and off. The states are represented by a pair of a configuration and a format of the root. Transitions are modelled by supercombinators. We formalise these notions when describing our Boolean encoding of CSP processes. In this section, we illustrate the two-level approach with a small example.

Example 1 *Let us consider the process* $R = P \underset{\{a\}}{||} Q[\![a/b]\!]$, *where* $P = a \rightarrow P \sqcap STOP$ *and* $Q = b \rightarrow Q$. *The process tree of R is presented on Figure 2(c). The explicit LTS machines underlying the semantics of the leaves P and Q are depicted on Figure 2(a) and Figure 2(b), respectively. The root process R contains a single format with two rules — one rule stating that if P performs* τ *then the entire system performs* τ *and another rule stating that if P performs a and Q performs b, R can perform a.*

2.1.4 Refinement Checking

Given two CSP processes *Spec* and *Impl*, the refinement check $Spec \sqsubseteq Impl$ reduces to checking for reverse containment of possible behaviours. For the traces model, $Spec \sqsubseteq_T Impl$ iff $traces(Impl) \subseteq traces(Spec)$.

We briefly outline how FDR carries out the refinement check. Let $OS_{Spec} = \langle S^s, s_0^s, L^s, T^s \rangle$ and $OS_{Impl} = \langle S^i, s_0^i, L^i, T^i \rangle$ be the labelled transition systems representing the operational semantics of *Spec* and *Impl*, respectively. As a preprocessing step, FDR normalises OS_{Spec}, so that OS_{Spec} reaches a unique state after any trace. The normalisation procedure requires as a precondition that OS_{Spec} be explicated and therefore *Spec* sequentialised. Essentially, the normalisation procedure transforms OS_{Spec} into the unique equivalent τ-free deterministic LTS with the fewest possible states (bisimulation-reduced). After normalising OS_{Spec}, FDR traverses the Cartesian product of OS_{Spec} and OS_{Impl} in a breadth-first manner, checking for compatibility of mutually-reachable states. For the traces model, a pair of states (s^s, s^i) is compatible, if $initials(s^i) \subseteq initials(s^s)$.

2.2 Bounded Model Checking

SAT-based bounded model checking [BCCZ99] is a symbolic model checking technique considered complementary to BDD-based model checking [BCM+92, McM93].

Bounded model checking focuses on searching for counterexamples of bounded length only.

The underlying idea is to fix a bound k and to unwind the model for k steps, thus considering behaviours and counterexamples of length at most k. In practice, BMC is conducted iteratively by progressively increasing k until either a counterexample is detected or k reaches a precomputed threshold called *completeness threshold* [CKOS05], which indicates that the model satisfies the specification. It is important to note that without knowing the completeness threshold, the BMC procedure is incomplete. Hence, BMC is mostly suitable for detecting bugs, not for verification (proving absence of bugs).

SAT-based BMC [BCCZ99] reduces the model checking problem to a propositional satisfiability problem. The idea is to construct a Boolean formula which is satisfiable if and only if there is a counterexample of length k. This formula is fed into a SAT-solver which decides the model checking problem in question and produces a counterexample, if any. Due to the DFS-nature of the SAT decision procedure, this technique allows for a fast detection of counterexamples.

3 Performing Bounded Trace Refinement

In this section, we present our iterative bounded refinement checking algorithm. Our approach for establishing trace refinement is based on watchdog transformations [RGM+03]. Our objective is the following. We are given two CSP processes *Spec* and *Impl* and an integer k. We aim at checking whether $Spec \sqsubseteq_T^k Impl$, i.e., whether all executions of the implementation of length at most k agree with the specification.

3.1 Preprocessing Phase Using FDR

Our implementation is intended as an alternative back-end for FDR, orthogonal to the standard explicit method of performing trace refinement. Currently, we use a shared library version of FDR for manipulating CSP processes and we mimic FDR up to the point of the final state-space exploration phase. Therefore, SymFDR reuses FDR's compiler and supercompiler and the data structures underlying the operational semantics.

At present, we use FDR to supercompile and normalise *Spec* and to retrieve LTS_{Spec} representing the operational semantics of *Spec*.

We assume that the implementation *Impl* comprises the interaction of c sequential processes $P_1, \ldots, P_c$ running in parallel, possibly using hiding and renaming. We write $Impl = P_1||P_2 \ldots ||P_c$ to denote a high-level process *Impl* with leaf processes $P_1, \ldots, P_c$. We use FDR to supercompile *Impl* and to retrieve both the set of supercombinators and the set $\{LTS_{P_i} | i \in \{1, \ldots, c\}\}$.

3.2 Watchdog Refinement Checking Algorithm

In a nutshell, the main steps of our algorithm are the following:

1. We transform *Spec* into a process *Watchdog* which allows the behaviours of both *Spec* and *Impl*. The transformation adds a special state *sink* to LTS_{Spec} and forces all erroneous traces (traces that do not conform with *Spec*) to be directed to *sink*.
2. We construct a process $Refinement = Watchdog \underset{\alpha_{Impl} \cup \alpha_{Spec}}{||} Impl =$ $Watchdog \underset{\alpha_{Impl} \cup \alpha_{Spec}}{||} (P_1||P_2 \ldots ||P_c)$

3. We check whether *Watchdog* can reach its *sink* state within k steps of the execution of *Refinement*.

The Watchdog Process. The transformation we apply on *Spec* is performed at the level of LTS_{Spec}. We add a state *sink* and make LTS_{Spec} total with respect to the alphabet $\alpha_{Spec} \cup \alpha_{Impl}$. The resulting process *Watchdog* operationally passes through *sink* whenever executing a trace that is not allowed by *Spec*. We allow an execution of *Watchdog* to contain any number of τ's after visiting *sink* in order to be able to increase the BMC bound by more than 1 at each step.

The Refinement Process. The process $Refinement = Watchdog \underset{\alpha_{Impl} \cup \alpha_{Spec}}{||} (P_1||P_2 \ldots ||P_c)$ can be used as an indicator whether *Impl* can behave in a way incompatible with *Spec*. *Watchdog* becomes just one of the sequential leaf processes of *Refinement*. It is evident then that:

1. $Spec \sqsubseteq_T Impl \Longleftrightarrow$ *Watchdog* never reaches its *sink* state in any execution of *Refinement*
2. All executions of *Refinement* forcing *Watchdog* to pass through its *sink* state constitute valid counterexamples of the assertion $Spec \sqsubseteq_T Impl$

4 Boolean Encoding of CSP Processes

In this section we present our encoding of CSP processes into Boolean formulae. First, we demonstrate how to encode sequential or explicated processes, corresponding to leaf processes in the operational representation. Then, we show how to glue together sequential processes with supercombinators to obtain an encoding of a high-level process. In what follows, we call a high-level process a concurrent system.

For the Boolean encoding we use the following notation. $\lceil X \rceil(Vars)$ denotes the Boolean encoding of X with respect to the vector(s) of Boolean variables *Vars*.

4.1 Encoding a Sequential Process

Let P be a finite-state process with alphabet of events Σ. Let $OS_P = \langle S, s_0, L = \Sigma^{\tau,\checkmark}, T \rangle$ be the LTS representing the operational semantics of P.

Encoding the Set of States. The basic idea is to enumerate the states in binary and represent them as Boolean functions. Each state $s \in S$ is identified by a bit vector $\overline{b} = (b_1, \ldots, b_n)$ of size $n = \lceil \log_2 |S| \rceil$ using an injective encoding $enc_S : S \to \{0,1\}^n$. We introduce an ordered vector of n distinct Boolean variables $\overline{x} = (x_1, \ldots, x_n)$. Each variable x_i uniquely identifies its corresponding bit b_i and, for each $s \in S$, $\lceil s \rceil(\overline{x})|_{\overline{x}=\overline{b}} = 1$ iff $enc_S(s) = \overline{b}$. We define $\lceil I \rceil(\overline{x}) = \lceil s_0 \rceil(\overline{x})$.

Encoding the Set of Labels. Using the same technique, we introduce an ordered vector $\overline{y} = (y_1, \ldots, y_m)$ of $m = \lceil \log_2 |L| \rceil$ distinct Boolean variables for encoding the set of labels $L = \Sigma^{\tau,\checkmark}$.

Encoding the Transition Relation. In order to represent the transition relation T, we introduce a copy $\overline{x'} = (x'_1, \ldots, x'_n)$ of $\overline{x} = (x_1, \ldots, x_n)$. $\overline{x}$ serves for representing the source states of transitions, $\overline{x'}$ – for representing the destination states. Then, for $t = (s_{src}, l, s_{dest}) \in T$, $\lceil t \rceil(\overline{x}, \overline{y}, \overline{x'}) = \lceil s_{src} \rceil(\overline{x}) \wedge \lceil l \rceil(\overline{y}) \wedge \lceil s_{dest} \rceil(\overline{x'})$. For any $s \in S$, we write $\lceil s \rceil(\overline{x'})$ to denote $\lceil s \rceil(\overline{x})[\overline{x'} \leftarrow \overline{x}]$, i.e. we represent s with respect to the variables $\overline{x}$ and then substitute the variables $\overline{x}$ with $\overline{x'}$. The encoding of the entire transition relation is the following: $\lceil T \rceil(\overline{x}, \overline{y}, \overline{x'}) = \bigvee_{t \in T} \lceil t \rceil(\overline{x}, \overline{y}, \overline{x'})$.

Encoding Executions. We can now represent a sequential process P implicitly by the pair of functions $\langle \lceil T^P \rceil(\overline{x},\overline{y},\overline{x'}), \lceil I^P \rceil(\overline{x}) \rangle$. For a given integer k, we define $Paths(P,k)$ to be the set of all executions $s_0 l_1 s_1 l_2 \ldots l_k s_k$ of OS_P of length k. If flattened to traces, $Paths(P,k)$ might contain traces of P of size less than k if τ's are present in the executions. In order to represent $Paths(P,k)$ symbolically, we introduce $(k+1)$ vectors of n Boolean variables $\overline{x_0},\overline{x_1}\ldots\overline{x_k}$ and k vectors of m Boolean variables $\overline{y_1},\overline{y_2}\ldots\overline{y_k}$. The vectors $\overline{x_0},\overline{x_1},\ldots,\overline{x_k}$ represent the states $s_0,s_1,\ldots,s_k$, respectively. Likewise, the vectors $\overline{y_1},\overline{y_2},\ldots,\overline{y_k}$ represent the labels of the corresponding transitions. Then $\lceil Paths(P,k) \rceil(\overline{x_0},\overline{x_1}\ldots\overline{x_k},\overline{y_1},\overline{y_2}\ldots\overline{y_k}) = \lceil I^P \rceil(\overline{x_0}) \wedge \bigwedge_{i=0}^{k-1} \lceil T^P \rceil(\overline{x_i},\overline{y_{i+1}},\overline{x_{i+1}})$.

4.2 Encoding a Concurrent System

Since a high-level root process can be modelled as an LTS, we now show how to encode a concurrent system similarly to a low-level sequential process.

A concurrent system is a set of processes running in parallel possibly using renaming and hiding. We denote by $Sys(c)$ the interaction of c sequential processes $P_1,...,P_c$ communicating over sets of events $\Sigma_1,....,\Sigma_c$, respectively. Let $\Sigma = \cup_{i=1}^{c} \Sigma_i$, $m = \lceil \log_2 |\Sigma^{\tau,\surd}| \rceil$.

Encoding the Sequential Processes. For $i \in \{1,...,c\}$, let $OP^i = \langle S^i, s_0^i, L^i = \Sigma_i^{\tau,\surd}, T^i \rangle$ be the LTS representing the operational semantics of P_i. Since $\Sigma_i \subseteq \Sigma$, we actually consider $L^i = \Sigma^{\tau,\surd}$.

For each process P_i, let $n_i = \lceil \log_2 |S^i| \rceil$. In order to represent S^i and the transition relation T^i, we introduce two copies of n_i Boolean variables $\overline{x^i} = (x_1^i,...,x_{n_i}^i)$ and $\overline{x^{i'}} = (x_1^{i'},...,x_{n_i}^{i'})$. The construction of $\lceil T^i \rceil(\overline{x^i},\overline{y^i},\overline{x^{i'}})$ and $\lceil I^i \rceil(\overline{x^i})$ follows the ideas from Section 4.1.

As illustrated in Section 4.1, for each process P_i we introduce a vector of m Boolean variables $\overline{y^i} = (y_1^i,...,y_m^i)$ for encoding the set $L^i = \Sigma^{\tau,\surd}$ symbolically. Thus, each process has its own set of variables for representing the alphabet $\Sigma^{\tau,\surd}$. We introduce an additional vector of Boolean variables $\overline{y} = (y_1,...,y_m)$ for encoding the resulting action of the entire system.

Encoding States (Configurations) of the Overall System. Recall that a concurrent system consists of multiple sequential processes running in parallel. A state of the entire system, also called a *configuration*, is identified by the current states of its sequential components. Formally, the set of states of the system is a c-ary relation $S \subseteq S^1 \times ... \times S^c$, the initial state being $s_0 = (s_0^1,...,s_0^c)$. Therefore, S can be represented symbolically using the Boolean variables from $\overline{x^1},...,\overline{x^c}$. If $s = (s^1,...,s^c) \in S$, then $\lceil s \rceil(\overline{x^1},...,\overline{x^c}) = \bigwedge_{i=1}^{c} (\lceil s^i \rceil(\overline{x^i}))$. For clarity, we denote the set of states of the overall system by *Configurations*.

Supercombinators and Formats. As we mentioned in Section 2.1.3, supercombinators are rules for combining together actions of the individual sequential processes into event-outcomes of the overall system [Ros98]. Within a supercombinator, each process can participate with a visible event, a silent action τ, or not be involved at all. We denote the non-involvement with the symbol ε. For any alphabet Σ, we let $\Sigma^{\varepsilon} = \Sigma \cup \{\varepsilon\}$. In addition, the set of supercombinators is partitioned into existing *formats*, i.e., different configurations of switched on and switched off processes among $P_1,\ldots,P_c$. We denote by *SC* the set of supercombinators and by *Formats* the set of formats of the concurrent system.

Formally, the set of supercombinators can be represented as a $(c+3)$-ary relation $SC \subseteq Formats \times \Sigma_1^{\tau,\surd,\varepsilon} \times ... \times \Sigma_c^{\tau,\surd,\varepsilon} \times \Sigma^{\tau,\surd} \times Formats$, or more generally $SC \subseteq Formats \times (\Sigma^{\tau,\surd,\varepsilon})^c \times \Sigma^{\tau,\surd} \times Formats$. $(f_{src}, a_1, ..., a_c, a, f_{dest}) \in SC$ iff from a certain configuration and a certain format

f_{src} of the overall system, P_1 performs a_1, ..., P_c performs a_c and the overall system performs a switching to a format f_{dest}.

The operational semantics of the concurrent system can be considered an implicit LTS, whose transitions can be switched on and off:

- set of states – $Formats \times Configurations$
- set of labels – SC
- transition relation – $T \subseteq (Formats \times Configurations) \times SC \times (Formats \times Configurations)$. If the system is in a given configuration and in a given format, the individual processes transition relations determine if the labels are switched on or off. Formally,
  $(f_i,(s_i^1,\ldots,s_i^c)) \overset{(f_i,a_1,\ldots,a_c,a,f_j)}{\rightarrow} (f_j,(s_j^1,\ldots,s_j^c))$ iff
  $(f_i,a_1,\ldots,a_c,a,f_j) \in SC \wedge \forall_{k=1}^{c}((a_k \neq \varepsilon \Rightarrow (s_i^k,a_k,s_j^k) \in T^k) \wedge (a_k == \varepsilon \Rightarrow s_i^k = s_j^k))$.

Encoding Supercombinators. For a given supercombinator $sc = (f_{src},a_1,...,a_c,a,f_{dest}) \in SC$, let $Passive(sc) = \{i \in \{1,\cdots,c\} | a_i = \varepsilon$, i.e. P_i is not involved in $sc\}$. Let $\overline{u} = (u_1,...,u_c)$ be a vector of (supercombinator-independent) Boolean variables. We denote:

$$lit(u_i) = \begin{cases} u_i & \text{if } P_i \text{ is not involved} \\ \neg u_i & \text{if } P_i \text{ performs a visible event or a } \tau \end{cases}$$

Note that a process might be switched on in a format and still be passive in a certain supercombinator in this format. Hence, we cannot use the format to conclude which processes are passive in a supercombinator.

Let $\overline{f}$ and $\overline{f'}$ be two vectors of $\lceil log_2|Formats|\rceil$ variables for encoding the source and destination format of a rule. Let $sc = (f_{src},a_1,...,a_c,a,f_{dest}) \in SC$. Then, $\lceil sc\rceil(\overline{y^1},...,\overline{y^c},\overline{y},\overline{u},\overline{f},\overline{f'}) = \bigwedge_{i \notin Passive(sc)}(\lceil a_i\rceil(\overline{y^i}) \wedge \neg u_i) \wedge \bigwedge_{i \in Passive(sc)} u_i \wedge \lceil a\rceil(\overline{y}) \wedge \lceil f_{src}\rceil(\overline{f}) \wedge \lceil f_{dest}\rceil(\overline{f'})$.

Hence, in an encoding of a supercombinator, we indicate a passive process P_i just by affirming a single Boolean variable u_i. We call u_i a *trigger*. For non-passive processes, we also encode the event that the process performs. The encoding of all supercombinators in all formats now becomes the following: $\lceil SC\rceil(\overline{y^1},...,\overline{y^c},\overline{y},\overline{u},\overline{f},\overline{f'}) = \bigvee_{sc \in SC}\lceil sc\rceil(\overline{y^1},...,\overline{y^c},\overline{y},\overline{u},\overline{f},\overline{f'})$.

Encoding a Transition of the Concurrent System. Let for $i \in \{1,\cdots,c\}$, $\psi_i(\overline{x^i},\overline{x^{i'}},\overline{y^i},u_i) :=$ if u_i then $(\overline{x^i} = \overline{x^{i'}})$ else $\lceil T^i\rceil(\overline{x^i},\overline{y^i},\overline{x^{i'}})$, where $\overline{x^i} = \overline{x^{i'}}$ is the short for $\bigwedge_{j=1}^{n_i}(x_j^i \Leftrightarrow x_j^{i'})$. The intuition behind a ψ_i is that, if P_i does not participate in a transition of the entire system, i.e. P_i is not involved in a supercombinator, P_i remains in the same state within its own labelled transition system OP^i. Otherwise, P_i progresses with respect to its transition relation T^i. Expressed as a Boolean formula, $\psi_i \equiv (u_i \wedge (\overline{x^i} = \overline{x^{i'}})) \vee (\neg u_i \wedge \lceil T^i\rceil(\overline{x^i},\overline{y^i},\overline{x^{i'}}))$.

We define a predicate $T^{Sys(c)}$ which is true exactly for the transitions of the overall system:

$$\begin{aligned}&\lceil T^{Sys(c)}\rceil(\overline{x^1},\cdots,\overline{x^c},\overline{x^{1'}},\cdots,\overline{x^{c'}},\overline{y^1},\cdots,\overline{y^c},\overline{y},\overline{u},\overline{f},\overline{f'}) = \\ &= \bigwedge_{i=1}^{c}\psi_i(\overline{x^i},\overline{x^{i'}},\overline{y^i},u_i) \wedge \lceil SC\rceil(\overline{y^1},\cdots,\overline{y^c},\overline{y},\overline{u},\overline{f},\overline{f'})\end{aligned}$$

Encoding Fixed Length Executions of the Concurrent System. Within the BMC framework, let k be the maximal bound for the length of the counterexamples we are looking for. Then:

$\lceil Paths(Sys(c),k)\rceil($
// variables for P_1 $\quad \overline{x_0^1},\ldots,\overline{x_k^1},\overline{y_1^1},\ldots,\overline{y_k^1},u_1^1,\ldots,u_k^1$
// variables for P_2 $\quad \overline{x_0^2},\ldots,\overline{x_k^2},\overline{y_1^2},\ldots,\overline{y_k^2},u_1^2,\ldots,u_k^2$
$\ldots$ $\quad \ldots$
// variables for P_c $\quad \overline{x_0^c},\ldots,\overline{x_k^c},\overline{y_1^c},\ldots,\overline{y_k^c},u_1^c,\ldots,u_k^c$
// variables for the traces of the system $\quad \overline{y_1},\ldots,\overline{y_k},$
// variables for the formats in the rules $\quad \overline{f_0},\ldots,\overline{f_k})$

$=$ // processes start from their initial states and the initial format is Format[0]
$\bigwedge_{j=1}^{c}\lceil I^j\rceil(\overline{x_0^j}) \ \wedge \lceil I^f\rceil(\overline{f_0}) \ \wedge$
// supercombinators as transitions at each of the k steps
$\bigwedge_{i=1}^{k}\lceil SC\rceil(\overline{y_i^1},\ldots,\overline{y_i^c},\overline{y_i},u_i^1,\ldots,u_i^c,\overline{f_{i-1}},\overline{f_i}) \ \wedge$
// the idea of the ψ formulas – either transitions or wait, depending on supercombinators
$\bigwedge_{\substack{j=1,\ldots,c\\ i=1,\ldots,k}}((u_i^j\wedge(\overline{x_{i-1}^j}=\overline{x_i^j}))\vee(\neg u_i^j\wedge\lceil T^j\rceil(\overline{x_{i-1}^j},\overline{y_i^j},\overline{x_i^j})))$

$=$
$\lceil I^{Sys(c)}\rceil(\overline{x_0^1},\ldots,\overline{x_0^c},\overline{f_0})\wedge$
$\bigwedge_{i=1}^{k}\lceil T^{Sys(c)}\rceil(\overline{x_{i-1}^1},\ldots,\overline{x_{i-1}^c},\overline{x_i^1},\ldots,\overline{x_i^c},\overline{y_i^1},\ldots,\overline{y_i^c},\overline{y_i},u_i^1,\ldots,u_i^c,\overline{f_{i-1}},\overline{f_i})$

5 Implementation Details

In the original version of BMC, the system is unwound step by step until the bound k is reached. Despite the recent advances in SAT-solvers' learning capabilities and incremental SAT-solving, we have observed that the bottleneck of the bounded refinement procedure is the SAT-solver. Therefore, we allow unfolding a configurable number i of steps of the process *Refinement* before running the SAT-solver. The SAT-solver is then used to check if *Refinement* can pass through the *sink* state in any of its last i unwindings. If yes, we have found a counterexample, otherwise we continue iterating until reaching the configured bound k. We refer to the value of i as *SAT-frequency*. We believe that this multi-step approach works well because the SAT-solver typically finds it much easier to find a satisfying assignment, if there is any, than to prove unsatisfiability, given CNF formulas with comparable size and structure.

We transform the Boolean formulae into equisatisfiable formulas in CNF using the Tseitin Encoding [BKWW08]. For brevity, we skip details about how we exploit the incremental SAT-interface. Currently, SymFDR supports MiniSAT (version 2.0), PicoSAT and ZChaff. For our test cases, we have found MiniSAT to be most efficient and all quoted results use MiniSAT. For our larger test cases, we also observed that MiniSAT finds a counterexample faster if we configure it to keep a smaller number of learned clauses and to restart more frequently. We also implemented adding unit learned clauses explicitly, as suggested in [ES03a]. Using positive polarity in decision heuristics also produced much better results.

The current implementation of SymFDR supports refinement checking systems with a single format only. However, we do not anticipate any problems generalising the problem to a multi-format setting. Moreover, most practical cases are also single-format.

In addition to the standard refinement check, SymFDR also supports the "Zig-Zag" temporal induction algorithm [ES03b], which makes BMC complete. However, due to concurrency, the

recurrence diameter is too big.

Some other approaches that did not scale well include exploiting unary rather than binary encoding, restricting the decision variables to the input ones [Sht00], incorporating PicoSAT's restarting scheme and phase saving strategy [Bie08] in MiniSAT, etc.

6 Experimental Results

In this section, we investigate the performance of SymFDR on a small number of case studies. We compare it to the performance of FDR, FDR used in a non-standard way, PAT [SLD08], and, in some cases, direct SAT encodings, NuSMV [CCG+02] and Alloy Analyzer [Jac06]. All SAT-based experiments use MiniSAT although SymFDR and the direct SAT encoder build upon MiniSAT version 2.0, while Alloy and NuSMV exploit the earlier version 1.14. All tests were performed on a 2.6 GHz PC with 2 MB RAM running Linux, except the test marked with a *, which was performed on a 4-MB-RAM PC running Linux, and the tests with PAT, which were performed on a 1.67 GHz PC with 2 MB RAM running Windows. The results are summarised in Table 1, Table 2 and Table 3. The last column titled $\sharp$ lists the length of counterexamples.

FDR-Div. The main search strategy for FDR is BFS [Ros94] because this has the combined advantages of always finding a shortest counterexample and of enabling implementations that work comparatively well on virtual memory. However, the strategy for discovering divergences is based on DFS. In test cases where it is likely that there are a good number of counterexamples, but that all of them occur comparatively deep in the BFS, there is good reason to use a bounded DFS (BDFS) algorithm to search for them, so that only error states reachable in less than some fixed number N of steps are reached. BDFS will quickly get to the depth where counterexamples are expected without needing to enumerate all of the levels where they are not. Provided that the counterexamples have something like a uniform distribution through the order in which the DFS discovers them, we can expect one to be found after searching through approximately $S/(C+1)$ states, where S is the total number of states and C is the number of counterexamples.

FDR does not implement such a strategy directly. It was, however, observed a number of years ago by Roscoe and James Heather that it is possible to use a trick that achieves the same effect using the present version of the tool. That is, arrange (perhaps using a watchdog) a system P' that performs only up to N events of the target implementation process P and then performs an infinite number of some indicator event when a trace specification is breached. Provided P is itself divergence-free, we then have that $P' \setminus \Sigma$ can diverge precisely when P violates the specification. FDR searches for this divergence by DFS.

This approach is particularly well suited to CSP codings of puzzles, since it is frequently known *ab initio* how long a counterexample will be, and the usual CSP coding uses the repeatable event *done* to indicate that the puzzle has been solved. The columns labelled FDR-Div in Table 1 and Table 2 report on the result of using this technique. In several ways this method is more similar to approach of PAT and SymFDR than the usual FDR approach. As is apparent from the experiments, there seems to be a large element of luck in how fast this approach is, possibly based on how close the path followed by the DFS is to a counterexample.

PAT. PAT [SLD08] is a model checker of a version of CSP enhanced with shared variables. Despite the BMC attempt [SLDS08], PAT is at present a fully explicit checker. In addition to LTL model checking, PAT supports CSP refinement checking which it performs in a way similar

to FDR although using DFS (instead of BFS), normalisation of the specification on-the-fly and partial-order reductions. In the test cases quoted here, the specification is given as a reachability property on the values of the shared variables. The reachability algorithm is based on DFS and state hashing is applied for compact state-space representation.

NuSMV. NuSMV [CCG+02] is a symbolic model checker verifying SMV against CTL properties using BDDs. The BMC framework of NuSMV, which we refer to as NuSMV-BMC, uses specifications written in LTL.

Alloy Analyzer. Alloy Analyzer [Jac06] is a fully-automatic tool for finding models of software systems designed in the lightweight Alloy modelling language. Alloy Analyzer could be considered a BMC checker due to its searching for a model only up to a certain scope and generating the model, if existing, using SAT-solving techniques.

Direct SAT Encodings. We believe that experimenting with direct SAT encodings of problems will offer guidance for optimising the translation of CSP to logic. For example, the chess knight test case suggests that a shorter chain of inference for high-level actions might be beneficial.

Test Cases. First, we consider the peg solitaire puzzle [Ros98], performing experiments on a chain of soluble boards with increasing level of difficulty. In the initial configuration, the board has all slots but one occupied by pegs. The only allowed move in the game is a peg hopping over another peg and landing on an empty slot. The hopped over peg is then removed from the board. The objective of the game is ending up with a board with a single peg positioned on the slot which had been initially empty. The length of any solution of the puzzle is equal exactly to the number N of pegs on the initial board — a hop event for $(N-1)$ pegs followed by an event *done* signifying a valid solution of the puzzle. The results are summarised in Table 1. The experiments indicate that for $N \geq 26$ SymFDR clearly outperforms FDR. In cases where a counterexample does not exist, FDR's BFS strategy outperforms the DFS-based tools PAT and SymFDR.

Our second test case is the chess knight tour. A knight is placed at position $(1,1)$ on an empty chess board of size $N \times N$. The objective is covering all squares of the board by visiting each square exactly once. Similarly to peg solitaire, a solution is generated as a counterexample to a specification asserting that the event *done* is never communicated. The length of a possible solution is N^2+1. The results are presented in Table 2. For $N=5$, FDR generates a counterexample faster, but for $N=6$ already, SymFDR found a solution in approximately 13 minutes, while FDR crashed after an hour and a half of state-space exploration.

The third test case — the classical puzzle of towers of Hanoi, aims primarily at comparing SymFDR with other SAT-based bounded checkers such as NuSMV and Alloy Analyzer. The results are summarised in Table 3. NuSMV-BMC and SymFDR seem to be competitive, both outperforming Alloy Analyzer. However, all non-SAT tools — FDR, PAT and NuSMV — are clearly orders of magnitudes more efficient than the SAT-based ones.

We can conclude that SymFDR is likely to outperform FDR in large combinatorial problems for which a solution exists, the length of the longest solution is relatively short (growing at most polynomially) and is predictable in advance. In those cases, we can fix the SAT-frequency close to a sizeable divisor of this length and thus spare large SAT overhead. The search space of those problems can be characterised as very wide (with respect to BFS), but relatively shallow — with counterexamples with length up to approximately 50–60. We suspect that problems with multiple solutions also induce good SAT performance. The experiments with the towers of Hanoi suggest that SAT-solving techniques offer advantages up to a certain threshold and weaken afterwards.

Table 1: Performance comparison – peg solitaire ($\sharp = N$)

N	FDR ♯ states checked	Time (sec.) FDR	FDR -Div	PAT	SymFDR SAT	SymFDR Total	SAT freq.	♯
20	41 703	0	0	6.14	6.92 10.82	10.23 14.06	10 20	20
23	411 976	5	0	2.16	11.21 6.62	16.72 12.33	12 23	23
26	4 048 216	72	0	7.23	27.73 15.39	35.16 24.66	13 26	26
29	28 249 254	581	1	89.93	54.29 39.33	65.39 49.61	15 29	29
32	$>$ 139 000 000 187 000 000*	$>$ 11 700 2 640*	5	8.91	175.05 172.56	189.20 186.61	16 32	32
35	—	—	1 485	399	529.88 291.59	548.80 309.94	18 35	35
38	—	—	43	1773.19	1 047.01	1 071.59	19	38
41	—	—	4	198.77	1 584.62	1 617.09	41	41

Table 2: Performance comparison – chess knight tour ($\sharp = N^2 + 1$)

N	FDR ♯ states checked	Time (sec.) FDR	FDR -Div	PAT	Direct SAT	SymFDR SAT	SymFDR Total	SAT freq.	♯
5	508 451	3	0.147	0.28	8.5	6.26 13.47	8.81 16.46	13 26	26
6	$>$ 120 000 000	$>$ 4 800	18	9.75	125.3	777.41	785.67	19	37
7	—	—	—	6.41	1 138	30 515.6	30 544.5	50	50

7 Conclusions and Future Work

In this paper we have demonstrated the feasibility of integrating a bounded refinement checker in FDR, and more specifically, exchanging the expensive explicit state-space traversal phase in FDR by a SAT check in SymFDR. On some test cases, such as complex combinatorial problems, SymFDR's performance is very encouraging, coping with problems that are beyond FDR's capabilities. In general, though, FDR usually outperforms SymFDR, particularly when a counterexample does not exist. We plan to further investigate and try to gain insight about the classes of problems that are tackled more successfully within the BMC framework.

We envision several directions for future work.

We plan to extend the BMC framework in SymFDR to make it applicable to the stable failures and failures-divergences models as well. This will involve extending the encoding of CSP processes with information about maximal refusals and divergences.

Table 3: Performance comparison – Hanoi towers ($\sharp = 2^N$)

N	Time (sec.)						SAT	$\sharp$
	FDR	PAT	NuSMV	SymFDR	Alloy	NuSMV-BMC	freq.	
5	0.198	0.64	0.43	4.92	11.53	2.157	16	32
6	0.202	2.89	0.66	27.26	327.37	34.910	32	64
7	0.164	5.01	0.171	182.68	21 537.27	1 864.75	64	128
8	0.182	27.76	0.292	3 114.05	—	2 218.23	128	256

We intend to implement McMillan's algorithm combining SAT and interpolation techniques to yield complete unbounded refinement checking [McM03]. This method has proven to be more efficient for positive BMC instances (instances with no counterexamples) than other SAT approaches. The completeness threshold in this case is the reverse depth of the state-space which is smaller than its recurrence diameter, as is the case with temporal induction [ES03b]. Moreover, experimental results have shown that, in practice, the algorithm often converges substantially faster, for bounds considerably smaller than the reverse depth. In addition, the interpolation algorithm allows jumping multiple time frames at once and hence allows tuning the SAT-frequency. The BMC framework presented in this paper will be the foundation to build upon.

Other avenues for further enhancing FDR's performance include partial-order reductions [Pel98] and CEGAR [COYC03, CCO+05].

Acknowledgements: We are grateful to D. Kroening and J. Worrell for their comments and P. Armstrong for his help with FDR. The analysis using DFS refinement through divergence checking was inspired by a correspondence several years ago between A. W. Roscoe and J. Heather. The work presented in this paper is supported by grants from EPSRC and US ONR.

Bibliography

[BCCZ99] A. Biere, A. Cimatti, E. M. Clarke, Y. Zhu. Symbolic Model Checking without BDDs. In *TACAS '99: Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*. Pp. 193–207. Springer-Verlag, London, UK, 1999.

[BCM+92] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, L. J. Hwang. Symbolic model checking: 1020 states and beyond. *Inf. Comput.* 98(2):142–170, 1992.

[Bie08] A. Biere. PicoSAT Essentials. *JSAT* 4(2-4):75–97, 2008.

[BKWW08] A. Biere, D. Kroening, G. Weissenbacher, C. Wintersteiger. *Digitaltechnik*. Springer, 2008.

[CCG+02] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, A. Tacchella. NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. In *Proc. International Conference on Computer-Aided Verification (CAV 2002)*. LNCS 2404. Springer, Copenhagen, Denmark, July 2002.

[CCO+05] S. Chaki, E. M. Clarke, J. Ouaknine, N. Sharygina, N. Sinha. Concurrent software verification with states, events, and deadlocks. *Formal Aspects of Computing* 17(4), 2005.

[CKOS05] E. Clarke, D. Kroening, J. Ouaknine, O. Strichman. Computational Challenges in Bounded Model Checking. *Software Tools for Technology Transfer (STTT)* 7(2):174–183, April 2005.

[COYC03] S. Chaki, J. Ouaknine, K. Yorav, E. M. Clarke. Automated compositional abstraction refinement for concurrent C programs: A two-level approach. In *Proc. SoftMC 03*. 2003.

[EB05] N. Een, A. Biere. Effective preprocessing in sat through variable and clause elimination. In *In proc. SAT05, volume 3569 of LNCS*. Pp. 61–75. Springer, 2005.

[ES03a] N. Een, N. Sorensson. An Extensible SAT-solver. In *SAT*. 2003.

[ES03b] N. Een, N. Sorensson. Temporal Induction by Incremental SAT-solving. In *Proceedings of First International Workshop on Bounded Model Checking*. ENTCS 4. 2003.

[G+05] M. Goldsmith et al. Failures-Divergence Refinement. FDR2 User Manual. Formal Systems (Europe) Ltd., June 2005.
doi:http://www.fsel.com/documentation/fdr2/fdr2manual.pdf

[Hoa85] C. A. R. Hoare. Communicating Sequential Processes. *Communications of the ACM* 21:666–677, 1985.

[Jac06] D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006.

[McM93] K. L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. PhD thesis, CMU, 1993.

[McM03] K. L. McMillan. Interpolation and SAT-Based Model Checking. In *CAV*. Pp. 1–13. 2003.

[Pel98] D. Peled. Ten Years of Partial Order Reduction. In *CAV '98: Proc. 10th International Conference on Computer Aided Verification*. Pp. 17–28. Springer-Verlag, London, UK, 1998.

[PY96] A. Parashkevov, J. Yantchev. ARC - a tool for efficient refinement and equivalence checking for CSP. In *IEEE 2nd International Conference on Algorithm and Architectures for Parallel Processing*. 1996.

[RGM+03] A. W. Roscoe, M. Goldsmith, N. Moffat, T. Whitworth, I. Zakiuddin. Watchdog transformations for property-oriented model checking. In *Proceedings of FME 2003*. 2003.
http://web.comlab.ox.ac.uk/oucl/work/bill.roscoe/publications/91.pdf

[Ros94] A. W. Roscoe. *Model-checking CSP*. Chapter 21. Prentice-Hall, 1994.
http://web.comlab.ox.ac.uk/oucl/work/bill.roscoe/publications/50.ps

[Ros98] A. W. Roscoe. *The theory and practice of concurrency*. Prentice Hall, 1998.
http://web.comlab.ox.ac.uk/oucl/work/bill.roscoe/publications/68b.pdf

[Ros08] A. W. Roscoe. Lecture notes for the course *Advanced Concurrency Tools*. Oxford University Computing Laboratory 2008.

[RRS+01] A. W. Roscoe, P. Ryan, S. Schneider, M. Goldsmith, G. Lowe. *The Modelling and Analysis of Security Protocols*. Addison-Wesley, 2001.

[Sht00] O. Shtrichman. Tuning SAT Checkers for Bounded Model Checking. In *CAV '00: Proceedings of the 12th International Conference on Computer Aided Verification*. Pp. 480–494. Springer-Verlag, London, UK, 2000.

[SLD08] J. Sun, Y. Liu, J. S. Dong. Model Checking CSP Revisited: Introducing a Process Analysis Toolkit. In *ISoLA*. Pp. 307–322. 2008.

[SLDS08] J. Sun, Y. Liu, J. S. Dong, J. Sun. Bounded Model Checking of Compositional Processes. In *Proceedings of the Second IEEE International Symposium on Theoretical Aspects of Software Engineering*. Pp. 23–30. IEEE, 2008.