SAT-Solving in CSP Trace Refinement

Hristina Palikareva, Joël Ouaknine and A. W. Roscoe

Department of Computer Science, Oxford University, UK

Abstract

In this paper, we address the problem of applying SAT-based bounded model checking (BMC) and temporal k-induction to asynchronous concurrent systems. We investigate refinement checking in the process-algebraic setting of Communicating Sequential Processes (CSP), focusing on the CSP traces model which is sufficient for verifying safety properties. We adapt the BMC framework to the context of CSP and the existing refinement checker FDR yielding bounded refinement checking which also lays the foundation for tailoring the k-induction technique. As refinement checking reduces to checking for reverse containment of possible behaviours, we exploit the SAT-solver to decide bounded language inclusion as opposed to bounded reachability of error states, as in most existing model checkers. Due to the harder problem to decide and the presence of invisible silent actions in process algebras, the original syntactic translation of BMC to SAT cannot be applied directly and we adopt a semantic translation algorithm based on watchdog transformations. We propose a Boolean encoding of CSP processes resting on FDR's hybrid two-level approach for calculating the operational semantics using supercombinators. We have implemented a prototype tool, SymFDR, written in C++, which uses FDR as a shared library for manipulating CSP processes and the state-of-the-art incremental SAT-solver MiniSAT 2.0. Experiments with BMC indicate that in some cases, especially in complex combinatorial problems, SymFDR significantly outperforms FDR and even copes with problems that are beyond FDR's capabilities. SymFDR in k-induction mode works reasonably well for small test cases, but is inefficient for larger ones as the threshold becomes too large, due to concurrency.

Keywords: CSP, FDR, concurrency, process algebra, SAT-solving, bounded model checking, k-induction, safety properties

1. Introduction

Model checking [CGP99; BK08; BHvMW09] is a powerful automatic formal verification technique for establishing correctness of systems. It requires a finite-state model of a system, capturing all its possible behaviours, and a specification property, usually modelled as a formula in some kind of temporal logic. The model checker performs analysis based on exhaustive exploration of the state space of the system to either confirm or refute that the system meets its specification. In the latter case, the model checker provides a counterexample trace for reproducing and fixing the bug. Model checking is complete and, therefore, reliable when pronouncing a system correct.

The main challenge in applying model checking in practice is the so-called *state-space explosion problem* which tends to be even more severe in the context of concurrent systems. The state-space of a concurrent system grows exponentially with the number of its concurrent components and the

number and size of its data values. This puts restrictions on the size of systems that can be tractably analysed.

To alleviate the state-space explosion problem, a significant number of techniques have been proposed. Methods for decreasing the size of the generated state space and enhancing the model checking algorithm include CEGAR [CGJ+00], partial-order reductions [CGP99; Pel98], bounded model checking [BCCZ99], etc. Regarding state-space representation, the major dichotomy is between explicit and symbolic [BCM+92; BCCZ99] model checking. *Explicit model checking* is based on explicit enumeration and examination of individual states. *Symbolic model checking* relies on abstract representation of sets of states, generally as Boolean formulae, and properties are validated using techniques such as BDD manipulation or SAT-solving.

The recent advances of efficient SAT-solvers have significantly broadened the horizons of symbolic model checking. SAT-based bounded model checking [BCCZ99] has proven to be an extremely powerful technique, mainly suited, due to its incompleteness, to falsification of properties. Approaches for making BMC complete include calculating completeness thresholds [CKOS04; CKOS05] or augmenting BMC with k-induction [SSS00; ES03b] or Craig interpolation [McM03] techniques.

Both bounded and unbounded SAT-based model checking have been mainly investigated in the context of hardware and sequential software systems. In this paper, we address applying BMC and temporal k-induction [ES03b] to asynchronous concurrent systems.

The general problem we investigate is *refinement checking* in process-algebraic settings and, more specifically, in the context of CSP [Hoa85; Ros98; Ros10].

In process algebras, systems are modelled as interactions of a collection of processes, communicating with each other and with the outer world via message passing, as opposed to shared variables. Using a high-level language, processes are defined compositionally and compiled into a hierarchical structure, starting with atomic process constructs and combining those using operators such as choice, parallel and sequential composition, hiding, etc. This allows for a way of describing reactive systems that is usually very concise and much more economical in state space than shared-variable languages.

Unlike in conventional model checking, where specifications are generally defined as temporal-logic formulae, in process algebras specifications are defined as abstract designs of the systems, i.e. as processes, which allows for a step-wise development process. The refinement checking procedure decides whether the behaviours of the system are a subset of the behaviours of the specification, i.e. whether the system refines the specification. Hence, the verification problem reduces to checking for reverse containment of behaviours and, therefore, to reverse language inclusion.

Developed in the late 1970's by Hoare, CSP is one of the two original process algebras. It allows for the precise description and analysis of event-based concurrency. An advantage of the CSP framework is that it offers a well-developed syntax, algebraic and operational semantics, a hierarchy of congruent denotational semantic models, as well as a formal theory of refinement and compositional verification. In terms of syntax and semantics, among other differences with existing formalisms for modelling concurrent systems, CSP supports the usage of broadcast communication, recursion, as well as hiding and renaming of events, both of which are powerful mechanisms for abstraction.

FDR [Ros94; FSE05] is acknowledged as the primary tool for CSP refinement checking and has been widely used for analysing safety-critical systems. The core of FDR is refinement checking in each of the semantic models, which is carried out on the level of the operational representation of the CSP processes and is implemented using explicit state enumeration supplemented by hierarchical state-space compression techniques. When deciding whether an 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. Although

until now FDR has followed the explicit model checking approach, there has been some work on the symbolic model checking of CSP resulting in the BDD-based refinement checker ARC [PY96] and the model checker PAT [SLDS08], both of which exploit a fully compositional encoding of CSP processes. PAT verifies systems defined in a version of CSP enhanced with shared variables and, within the BMC framework, it uses specifications defined as reachability properties on the values of the shared variables, which requires a different model checking algorithm based on reachability and not on language containment.

This paper reports our attempts to integrate SAT-based BMC and temporal k-induction into FDR. The former technique is incomplete and as such is only suitable to detecting bugs. k-induction, however, is complete and can also be used for establishing the correctness of systems. Hence, to the best of our knowledge, this is the first attempt to apply unbounded SAT-based refinement checking to CSP. We propose an alternative Boolean encoding of CSP processes based on FDR's hybrid two-level approach for calculating the operational semantics using supercombinators [Gol04]. As we deal with with a problem that reduces to language inclusion instead of to reachability and due to the presence of invisible hidden actions in process algebras, the original syntactic translation of BMC to SAT cannot be applied directly and we adopt a semantic translation algorithm based on watchdog transformations [RGM$^+$03]. Essentially, this involves reducing a refinement check into analysing a single process which is constructed by putting the implementation process in parallel with a transformed specification process. The latter plays the role of a watchdog that monitors and marks violating behaviours. Within the scope of this paper, we only consider the translation of *trace* refinement to SAT checking.

The result is a prototype tool SymFDR[1] which, when combined with state-of-the-art SAT-solvers such as MiniSAT 2.0 [ES03a; EB05], sometimes outperforms FDR by a significant margin in 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 tailored to the specific 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 SAT-based BMC and k-induction. In Section 3, we show how to adapt the watchdog approach [RGM$^+$03] to BMC and, hence, to k-induction, 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 evaluation.

2. Preliminaries

2.1. CSP and FDR

In this section, we give a brief overview of CSP and FDR. The interested reader is referred to [Ros98], [Ros10] and [FSE05] for more details. We restrict our focus to the traces model exclusively, intentionally omitting information about other more expressive models of CSP.

2.1.1. CSP Syntax

In CSP, processes interact with each other and an external environment by communicating events. More than one process may have to cooperate in the performance of an event, i.e. handshake on it. It

[1] SymFDR's binaries are expected to be online shortly.

is standard to distinguish between visible events that might need the cooperation of other processes or the environment and invisible internal actions that occur silently, are not observable or controllable outside a process and model an internal computation such as nondeterminism, unfolding of a recursion, abstraction of details.

Let Σ be a finite alphabet of visible events with $\tau, \checkmark \notin \Sigma$. τ denotes the invisible silent action and $\checkmark$ — a successful termination of a process — a special action which is visible but uncontrollable from outside and can only occur last. 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. For a given process P, we denote by $\alpha_P \subseteq \Sigma^{\checkmark}$ the set of all visible events that P can perform. We recall the core syntax of CSP.

Definition 2.1. *A CSP process is defined recursively via the following grammar:*

$$P \mathrel{\widehat{=}} \quad STOP \mid SKIP \mid DIV \mid x : A \longrightarrow P(x) \mid P_1 \,\Box\, P_2 \mid P_1 \sqcap P_2 \mid \\ P_1 \underset{B}{\parallel} P_2 \mid P_1 \fatsemi P_2 \mid P \setminus A \mid P[\![R]\!] \mid \mu\, P \bullet F(P)$$

STOP represents a deadlocked process, i.e. a process that is not capable of communicating any visible or τ actions. The prefixed process $x : A \longrightarrow P(x)$ initially offers the environment to perform any event a from A and subsequently behaves like $P(a)$. *DIV* denotes a livelock, i.e. a process that is engaged in performing an infinite loop of internal τ actions without ever communicating with the external environment. The process *SKIP* denotes successful termination and is willing to perform $\checkmark$ at any time. $P_1 \,\Box\, P_2$ and $P_1 \sqcap P_2$ denote, respectively, external and internal choice of P_1 and P_2. In the former case the choice is resolved by the environment while in the latter — nondeterministically. The parallel composition $P_1 \underset{B}{\parallel} P_2$ can communicate an event from B only if both P_1 and P_2 are ready to do so — P_1 and P_2 need to handshake (synchronise) on the events in B, but can perform independently on all other events. In practice, it is common to synchronise P_1 and P_2 on their shared events, i.e. use $B = \alpha_{P_1} \cap \alpha_{P_2}$. The sequential composition $P_1 \fatsemi P_2$ behaves like P_1 until P_1 terminates successfully, at which point it silently evolves into P_2. $P \setminus A$ behaves like P except that all events from A are being hidden, i.e. turned into internal τ actions. Hence, the A events in P become invisible and uncontrollable by other processes or the environment by means of synchronisation. $P[\![R]\!]$ behaves like P, except that, whenever P can perform an event a, $P[\![R]\!]$ can perform any event b, such that aRb. $\mu\, P \bullet F(P)$ denotes a recursive process.

FDR supports the language CSP_M which extends core CSP with several further operators and an extensive functional language. Our prototype tool SymFDR supports the full CSP_M syntax, except that it cannot at present handle scripts using the function *chase*.

2.1.2. Running Example: Milner's Scheduler

Given a fixed $N \in \mathbb{N}$, a scheduler must arrange two classes of events $a.i$ and $b.i$ for $i \in \{0, \ldots, N-1\}$. There are two requirements — the $a.i$-s should occur in strict rotation, i.e. $\underbrace{a.0, a.1, \ldots, a.N-1}$, $\underbrace{a.0, a.1 \ldots, a.N-1}, \ldots$, and there should be precisely one $b.i$ between each pair of $a.i$-s.

In CSP, Milner's scheduler can be modelled as a ring of cell processes synchronised using extra events $c.i$, as illustrated on Figure 1. An abstracted CSP script for Milner's scheduler establishing the rotation specification is presented on Figure 2. $i \oplus 1$ denotes $(i+1)\%N$ and, likewise, $i \ominus 1$ — $(i-1)\%N$. For each process $Cell(i)$, we extend its alphabet $\alpha_{Cell(i)} = \{a.i, b.i\}$ with extra events $c.i$ and $c.i \oplus 1$ to use for synchronising with its neighbouring processes $Cell(i \ominus 1)$ and $Cell(i \oplus 1)$, respectively. $Cell(0)$ is defined in a slightly different way as the a-sequence should start with $a.0$.

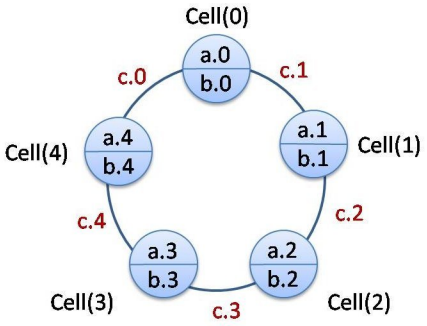

Figure 1: Milner's scheduler for $N = 5$

The scheduler is constructed by composing all the cells in parallel and hiding all c-events on top as they have been introduced solely for synchronisation purposes. Within *Scheduler*, the operator $\|$ corresponds to synchronising the two argument processes on the set of their common events and is fully associative. Hence, a cell can perform an event only if all other cells that have the same event in their alphabet are also offering to do so. $\{|\, c \,|\}$ is a shorthand for $\{c.i \mid i \in \mathbb{N}\}$.

We give a rough idea of how *Scheduler* works and why it preserves the rotation specification (modelled as *Spec*). The only process that can initially perform an event is $Cell(0)$ — for all $i > 0$, $Cell(i)$ is blocked as it needs $Cell(i-1)$ to also offer $c.i$. After $Cell(0)$ communicates $a.0$, the only thing that can happen next is $Cell(0)$ and $Cell(1)$ synchronising on $c.1$, thus enabling $Cell(1)$ to perform $a.1$. Concerning the sequence of a-s, the same process is repeated around the ring as, synchronising on $c.i \oplus 1$, $Cell(i)$ passes a token to $Cell(i \oplus 1)$ to signify that it is $Cell(i \oplus 1)$'s turn to contribute an $a.i \oplus 1$. Obviously, the second requirement for the scheduler is captured as well, also in the most general way.

$$
\begin{array}{l}
Cell(0) = a.0 \longrightarrow c.1 \longrightarrow b.0 \longrightarrow c.0 \longrightarrow Cell(0) \\
Cell(i) = c.i \longrightarrow a.i \longrightarrow c.i \oplus 1 \longrightarrow b.i \longrightarrow Cell(i) \quad \text{if } i > 0 \\
Scheduler = (Cell(0) \parallel Cell(1) \parallel \ldots \parallel Cell(N-1)) \setminus \{|\, c \,|\} \\
ASpec(i) = a.i \longrightarrow ASpec(i \oplus 1) \\
Spec = ASpec(0) \\
\text{assert } Spec \sqsubseteq_T Scheduler \setminus \{|\, b \,|\}
\end{array}
$$

Figure 2: CSP syntax — Milner's scheduler

2.1.3. Denotational Semantics

Traditionally, the primary means of understanding CSP descriptions has been to use denotational (behavioural) models, whereby a process is identified with the set of observable behaviours it can exhibit.

CSP supports a hierarchy of several such denotational semantic models. Different models describe different types of behaviours, providing more or less information about a process, with the natural trade-off between the amount of details recorded for a process and the complexity of working in the model. All denotational models are compositional in the sense that the denotational value (the set of possible behaviours) of each process can be computed in terms of the denotational values of its

subcomponents. The value of a recursive process is obtained using standard fixed-point theory in the style of Scott and Strachey.

In the simplest of all models, the traces model, a process P is identified with the set of its finite traces, denoted by $\mathsf{traces}(P)$. Intuitively, a trace of a process is a sequence of visible actions that the process can perform. Naturally, the set of traces of a process is non-empty and prefix-closed. For example, going back to the scheduler described on Figure 2, a trace of $Cell(0)$ is any prefix of $(a.0\ c.1\ b.0\ c.0)^*$. The traces model records information about what a process may do and is, therefore, sufficient for verifying safety properties.

There are two different approaches for obtaining the set $\mathsf{traces}(P)$ — either by constructing it inductively from the traces of its subcomponents, or by extracting it from the operational representation. Refer to [Ros98] for the rules underlying the first approach. Just to give a flavour of it in terms of Milner's scheduler:

- $\mathsf{traces}(Spec) = \{\langle\rangle\} \cup \{\langle a.0\rangle^\frown t \mid t \in \mathsf{traces}(ASpec(1))\}$,
- $\mathsf{traces}(Scheduler) = \{t \upharpoonright (\Sigma \setminus \{|\, c \,|\}) \mid t \in \mathsf{traces}(Cell(0) \ \|\ \ldots \ \|\ \ Cell(N-1))\}$.

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 standard congruence theorems, presented and proven in [Ros98; Ros88].

2.1.4. Operational Semantics

The operational semantics models CSP processes as labelled transition systems (LTS's), with nodes denoting processes and labels denoting visible events 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 can generally be calculated by repeatedly applying a set of SOS-style 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 firing rules are presented below. We use an auxiliary process term Ω to denote any process that has already terminated successfully. If F is a CSP term with a free process variable X and Q is a CSP process, $F[Q/X]$ represents the process obtained by substituting every free occurrence of X in F with Q. The last three rules reflect the fact that termination is distributive — $P_1 \underset{A}{\parallel} P_2$ terminates when both P_1 and P_2 do. The reader is referred to [Ros98] for more information.

$$\frac{}{SKIP \xrightarrow{\checkmark} \Omega} \qquad (SKIP)$$

$$\frac{}{x : A \to P(x) \xrightarrow{a} P(a)}(a \in A) \qquad (\xrightarrow{a})$$

$$\frac{}{P_1 \sqcap P_2 \xrightarrow{\tau} P_1} \qquad \frac{}{P_1 \sqcap P_2 \xrightarrow{\tau} P_2} \qquad (\sqcap)$$

$$\frac{}{\mu\, P \bullet F(P) \xrightarrow{\tau} F[(\mu\, P \bullet F(P))/P]} \qquad (\mu\, P \bullet F(P))$$

$$\frac{P_1 \xrightarrow{\tau} P_1'}{P_1 \,\Box\, P_2 \xrightarrow{\tau} P_1' \,\Box\, P_2} \qquad \frac{P_2 \xrightarrow{\tau} P_2'}{P_1 \,\Box\, P_2 \xrightarrow{\tau} P_1 \,\Box\, P_2'} \qquad (\Box_\tau)$$

$$\frac{P_1 \xrightarrow{b} P_1'}{P_1 \,\Box\, P_2 \xrightarrow{b} P_1'}(b \in \Sigma^{\checkmark}) \qquad \frac{P_2 \xrightarrow{b} P_2'}{P_1 \,\Box\, P_2 \xrightarrow{b} P_2'}(b \in \Sigma^{\checkmark}) \tag{$\Box_{\Sigma^{\checkmark}}$}$$

$$\frac{P_1 \xrightarrow{\tau} P_1'}{P_1 \fatsemi P_2 \xrightarrow{\tau} P_1' \fatsemi P_2} \tag{$\fatsemi_{\tau}$}$$

$$\frac{P_1 \xrightarrow{a} P_1'}{P_1; P_2 \xrightarrow{a} P_1'; P_2}(a \in \Sigma) \qquad \frac{\exists\, P_1'.P_1 \xrightarrow{\checkmark} P_1'}{P_1; P_2 \xrightarrow{\tau} P_2} \tag{$\fatsemi_{\Sigma^{\checkmark}}$}$$

$$\frac{P \xrightarrow{\tau} P'}{P \setminus A \xrightarrow{\tau} P' \setminus A} \tag{$\setminus A_{\tau}$}$$

$$\frac{P \xrightarrow{\checkmark} P'}{P \setminus A \xrightarrow{\checkmark} \Omega} \tag{$\setminus A_{\checkmark}$}$$

$$\frac{P \xrightarrow{a} P'}{P \setminus A \xrightarrow{\tau} P' \setminus A}(a \in A) \qquad \frac{P \xrightarrow{a} P'}{P \setminus A \xrightarrow{a} P' \setminus A}(a \in \Sigma \setminus A) \tag{$\setminus A_{\Sigma}$}$$

$$\frac{P \xrightarrow{\tau} P'}{P[\![R]\!] \xrightarrow{\tau} P'[\![R]\!]} \tag{$[\![R]\!]_{\tau}$}$$

$$\frac{P \xrightarrow{\checkmark} P'}{P[\![R]\!] \xrightarrow{\checkmark} \Omega} \qquad \frac{P \xrightarrow{a} P'}{P[\![R]\!] \xrightarrow{b} P'[\![R]\!]}(aRb) \tag{$[\![R]\!]_{\Sigma^{\checkmark}}$}$$

$$\frac{P_1 \xrightarrow{\tau} P_1'}{P_1 \underset{A}{\parallel} P_2 \xrightarrow{\tau} P_1' \underset{A}{\parallel} P_2} \qquad \frac{P_2 \xrightarrow{\tau} P_2'}{P_1 \underset{A}{\parallel} P_2 \xrightarrow{\tau} P_1 \underset{A}{\parallel} P_2'} \tag{$\underset{A}{\parallel}_{\tau}$}$$

$$\frac{P_1 \xrightarrow{a} P_1'}{P_1 \underset{A}{\parallel} P_2 \xrightarrow{a} P_1' \underset{A}{\parallel} P_2}(a \in \Sigma \setminus A) \qquad \frac{P_2 \xrightarrow{a} P_2'}{P_1 \underset{A}{\parallel} P_2 \xrightarrow{a} P_1 \underset{A}{\parallel} P_2'}(a \in \Sigma \setminus A) \tag{$\underset{A}{\parallel}_{\Sigma_1}$}$$

$$\frac{P_1 \xrightarrow{a} P_1',\ P_2 \xrightarrow{a} P_2'}{P_1 \underset{A}{\parallel} P_2 \xrightarrow{a} P_1' \underset{A}{\parallel} P_2'}(a \in A) \tag{$\underset{A}{\parallel}_{\Sigma_2}$}$$

$$\frac{P_1 \xrightarrow{\checkmark} P_1'}{P_1 \underset{A}{\parallel} P_2 \xrightarrow{\tau} \Omega \underset{A}{\parallel} P_2} \qquad \frac{P_2 \xrightarrow{\checkmark} P_2'}{P_1 \underset{A}{\parallel} P_2 \xrightarrow{\tau} P_1 \underset{A}{\parallel} \Omega} \tag{$\underset{A}{\parallel}_{\checkmark_1}$}$$

$$\frac{}{\Omega \underset{A}{\parallel} \Omega \xrightarrow{\checkmark} \Omega} \tag{$\underset{A}{\parallel}_{\checkmark_2}$}$$

Extracting Behaviours from Operational Semantics. We now present how behaviours, in our case – traces, can be retrieved from the operational semantics of a process. Intuitively, a trace of a process is obtained by trimming the invisible τ actions from an execution of the LTS underlying the operational representation.

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 $s \xrightarrow{l} s'$. An execution of M is a finite or 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 $i \geq 0$, $s_i \xrightarrow{l_{i+1}} s_{i+1}$. Going back to our example, the LTS's of *Cell(0)* and *Cell(1)* are depicted on Figure 3(b).

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 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:

- $\mathsf{initials}(p) = \{l \in \Sigma^\checkmark \mid p \xrightarrow{l}\}$, i.e. $\mathsf{initials}(p)$ is the set of visible events that can be communicated from the state p.
- For $t = \langle x_i \mid 0 \leq i < n\rangle \in (\Sigma^\tau)^{*\checkmark}$, 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 $p \overset{t}{\Longrightarrow} q$ 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 $\mathsf{traces}(P) = \{t \in \Sigma^{*\checkmark} \mid \exists\, q \in S^P . s_0^P \overset{t}{\Longrightarrow} q\}$.

2.1.5. Operational Representation in FDR — The Two-Level Approach

The SOS notation for operational semantics allows the creation of many operators that do not fit in the denotational world of CSP. Any CSP operator can be described using less general *combinator* operational rules instead and, conversely, any operator that can be given combinator operational semantics can be derived and given denotational semantics in CSP [Ros10]. Combinator-style semantics can be generalised to *supercombinator* operational semantics which is the one used in FDR. We give details about both combinator and supercombinator semantics below.

Combinator Rules. As with SOS, there are several combinator rules for each CSP operator and these allow us to infer the initial actions available at each process node out of its top-level operator and the initial actions available at its immediate process arguments. The crucial difference compared to SOS rules originates from the fact that process arguments can be viewed as switched on or off, depending on the context they are used in. Given a compound CSP process $P = \otimes(P_1, \ldots, P_n)$, a process argument P_i is considered *switched on* if its initial actions are immediately relevant for the initial actions of P and *switched off* if $\otimes$ does not need its initial actions to deduce the resulting initial action of P. For example:

- in $P_1 \mid\mid\mid P_2 = P_1 \underset{\emptyset}{\parallel} P_2$, both P_1 and P_2 are switched on
- in $P_1 \fatsemi P_2$, P_2 is initially switched off until P_1 performs $\checkmark$, at which point P_1 becomes switched off and P_2 switched on
- in $a \longrightarrow P$, P is initially switched off but gets switched on when a is communicated

- in $P_1 \sqcap P_2$, P_1 and P_2 are initially switched off as the nondeterministic choice is only resolved after a τ is performed, at which point precisely one of the two processes becomes activated.

Combinator rules keep track of which processes are switched on at every given moment and restrict SOS by allowing only two types of rules:

- rules enforcing that whenever a switched-on process argument performs a τ, this is promoted to a τ of the compound process that does not change its structure.
- rules combining visible events of switched-on process arguments (if any) into a resulting action of the compound process. In those rules, a switched-on process can participate with either a visible event or not be involved at all, the latter of which we denote with the symbol ϵ.

Combinator rules also need to indicate the structure of the successor term. In many instances, the structure is the same as the initial one and so does not have to be mentioned explicitly in the rules. When the structure does change (i.e. processes become switched from on to off or conversely), this is indicated by a CSP term in which the various arguments of the operator may appear. In every case, the successor state contains the original argument if the latter has not participated in the action, or the state that the argument has moved to if it did.

Now formally [Ros11], let P be a compound process with a top-level operator $\otimes$, switched-on arguments $P_1, \ldots, P_n$ (for some $n \geq 0$) and switched-off arguments $Q = \langle P_\lambda \mid \lambda \in \Lambda \rangle$.

Having any switched-on process argument P_i that can go via a τ to a state P_i', the τ-promotion rule takes the form:

$$\otimes(P_1, \ldots, P_i, \ldots P_n, Q) \stackrel{\tau}{\longmapsto} \otimes(P_1, \ldots, P_i', \ldots P_n, Q).$$

As this rule holds universally for any switched-on argument of any CSP operator, τ-promotion rules do not need to be added explicitly to the combinator operational semantics as they were in the SOS rules $\Box_\tau, \fatsemi_\tau, \backslash A_\tau, [\![R]\!]_\tau, \parallel_{A\ \tau}$.

Rules combining visible events take the general form $((x_1, \ldots, x_n), y, T)$, where $x_i \in \Sigma^\checkmark \cup \{\epsilon\}$, $y \in \Sigma^{\tau,\checkmark}$ and T is a piece of CSP syntax specifying the structure of the successor term. The idea is that whenever all P_i's that have $x_i \neq \epsilon$ can perform x_i and go to states P_i', they can synchronise to make the compound process P perform y and enter a state T. The successor state T is either Ω, if $y = \checkmark$, or is specified by an open CSP term in which the free variables are indices drawn from $\{1, \ldots, n\} \cup \{\text{-}\lambda \mid \lambda \in \Lambda\}$, which get substituted according to the following rules:

- For $i \in \{1, \ldots, n\}$, we distinguish different cases. If $x_i = \epsilon$ or $x_i \in \Sigma$, i is replaced by P_i or P_i', respectively. If $x_i = \checkmark$, i does not appear in the successor term T any more as P_i becomes switched off.
- An index $\text{-}\lambda$ for $\lambda \in \Lambda$ indicates that the process P_λ has become switched on and is replaced by P_λ.

We list the combinator rules below. In some of them, e.g. $\fatsemi_\Sigma\ \backslash A_\Sigma$, $[\![R]\!]_\Sigma$, $\parallel_{A\ \checkmark_1}$ and $\parallel_{A\ \checkmark_2}$, the structure of the successor term does not change, i.e. the resulting state is $\otimes(P_1'', \ldots, P_n'', Q)$, where $P_i'' = P_i$ if $x_i = \epsilon$ and $P_i'' = P_i'$ if $x_i \in \Sigma$. In those cases, we omit T from the rules for simplicity. In rules *SKIP*, $\stackrel{a}{\longrightarrow}$ and $\sqcap$, there is no switched-on argument initially which we indicate by $-$. Ω is naturally switched off as it represents successful termination. Hence, $\Omega \parallel_A$ and $\parallel_A \Omega$ are viewed as unary operators.

$$((-), \checkmark, \Omega) \qquad (SKIP)$$

$$((-), a, \text{-}1) \qquad (\xrightarrow{a})$$

$$((-), \tau, \text{-}1) \text{ and } ((-), \tau, \text{-}2) \qquad (\sqcap)$$

$$((\checkmark, \epsilon), \checkmark, \Omega) \text{ and } ((\epsilon, \checkmark), \checkmark, \Omega) \qquad (\Box_{\checkmark})$$

$$((a, \epsilon), a, 1) \text{ and } ((\epsilon, a), a, 2) \qquad (\Box_{\Sigma})$$

$$((a), a) \text{ and } ((\checkmark), \tau, \text{-}2) \qquad (\fatsemi_{\Sigma\checkmark})$$

$$((\checkmark), \checkmark, \Omega) \qquad (\backslash A_{\checkmark})$$

$$((a), \tau) \text{ if } a \in A \text{ and } ((a), a) \text{ if } a \in \Sigma \setminus A \qquad (\backslash A_{\Sigma})$$

$$((\checkmark), \checkmark, \Omega) \text{ and } ((a), b) \text{ when } aRb \qquad ([\![R]\!]_{\Sigma\checkmark})$$

$$((a, \epsilon), a) \text{ and } ((\epsilon, a), a) \text{ if } a \in \Sigma \setminus A \qquad (\underset{A}{\parallel}_{\Sigma_1})$$

$$((a, a), a) \text{ if } a \in A \qquad (\underset{A}{\parallel}_{\Sigma_2})$$

$$((\checkmark, \epsilon), \tau, \Omega \underset{A}{\parallel} 2) \text{ and } ((\epsilon, \checkmark), \tau, 1 \underset{A}{\parallel} \Omega) \qquad (\underset{A}{\parallel}_{\checkmark_1})$$

$$((a), a) \text{ if } a \in \Sigma \setminus A \text{ and } ((\checkmark), \tau, SKIP) \qquad (\Omega \underset{A}{\parallel}, \underset{A}{\parallel} \Omega)$$

Supercombinator Operational Semantics. Combinator operational semantics captures precisely CSP-definable operators [Ros10; Ros11]. However, actions of compound processes need to be calculated recursively on-the-fly out of the actions of their subterms. Furthermore, successor states are presented as pieces of syntax which does not prove to be efficient when analysing large systems.

Supercombinator operational semantics is less general but more efficient version of the combinator operational semantics [Ros10]. Supercombinator rules take the form of combinator ones, but are generalised to combine together actions of subprocesses nested under an *arbitrary* number of applications of CSP operators. As there is no combinator rule for recursion, the only constraint is that any process argument should be a closed CSP term, i.e. should have all the recursion unwound. Based on this assumption, all process arguments have combinator semantic rules which can be composed together to obtain rules for the outermost CSP operator. Hence, it transforms a combination of CSP operators into a single one. Furthermore, this can be implemented efficiently in a single run before the state-space exploration phase rather than on-the-fly when needed using recursive calls.

We illustrate the approach by example. Let $P = \otimes_1(\otimes_2(P_1, P_2), \otimes_3(P_3, P_4))$ and let us assume for simplicity that all $\otimes_1, \otimes_2$ and $\otimes_3$ have their two arguments switched on and their application does not result in successor terms with a different structure (format). Considering the τ-promotion rules, if $\otimes_2$ or $\otimes_3$ have a rule that generates a τ, this τ gets promoted by $\otimes_1$. For instance, if $\otimes_2$ has the rule $((a, b), \tau)$, then we create a supercombinator rule $((a, b, \epsilon, \epsilon), \tau)$ for the compound process $\otimes_1$. The other type of supercombinator arises when we can match all input requirements of one of $\otimes_1$'s combinators using combinators of $\otimes_2$ and $\otimes_3$ that produce visible results. For example, if $\otimes_1, \otimes_2$ and $\otimes_3$ have the rules $((a, b), c)$, $((\epsilon, a), a)$ and $((b, d), b)$, respectively, then the composition will have $((\epsilon, a, b, d), c)$.

Supercompiling — the process of associating supercombinator-style operational semantics to a CSP process, follows a hybrid high-/low-level approach for calculation and representation [Ros10]. It identifies all true recursions and compiles them on a low level, generating explicit LTS's using the combinator rules. What remains for the high level are closed processes combined typically using parallel composition, hiding and renaming, although the dividing line is somewhat more complex and is drawn where sensible. For example, the choice operators and sequential composition can also be lifted to the high level as long as their arguments are all closed terms.

The result of supercompilation is a high-level structure which consists of two parts. The first one is a process tree with leaves – low-level compiled LTS's, and internal nodes – CSP operators compiled on the high level, usually hiding, renaming or parallel composition. Each node, even if internal, represents a process and can be interpreted as an LTS with its behaviours deducible on-the-fly from the behaviours of its children. The second part of the high-level structure is a set of supercombinators mapping actions of a number of leaf processes to an event-outcome of the composite root process [Ros98]. In what follows, we use the notions of supercombinators and rules interchangeably. We note that the list of leaf processes together with the set of supercombinators is a complete characterisation of the high-level process as the semantics of all CSP operators corresponding to the internal nodes in the process tree is captured by the supercombinator rules. The structured process tree can be used, though, for making the whole high-level process completely explicit.

The set of combinators is partitioned with respect to the existing *formats* — the different configurations of switched on and switched off leaf processes. In the worst case, the number of formats can be exponential in the number of leaves, but in practice this is rarely the case and quite often, there is just a single format, especially when composing processes in parallel on the top level.

Within a supercombinator, each process can participate with a visible event, a silent action τ, or not be involved at all, the latter of which we again denote by ϵ. As with combinator rules, the supercompiler generates two types of rules[Ros98; Gol04; RRS+01]:

- a rule for a leaf process 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 $P = a \longrightarrow P$ and $Q = b \longrightarrow Q$, then if P performs a and Q performs b, $P \underset{\{a\}}{\parallel} Q[\![\mathrm{a}/b]\!]$ can perform a, where $Q[\![\mathrm{a}/b]\!]$ is the process Q with the event b being renamed to a. Hence, $((a, b), a)$ is a valid rule for the root process $P \underset{\{a\}}{\parallel} Q[\![\mathrm{a}/b]\!]$ with leaves P and Q.

Going back to our running example, after supercompiling *Scheduler* $\setminus \{|\, b \,|\}$, we obtain the process tree depicted on Figure 3(a). The simple recursive cell processes are compiled as leaves and

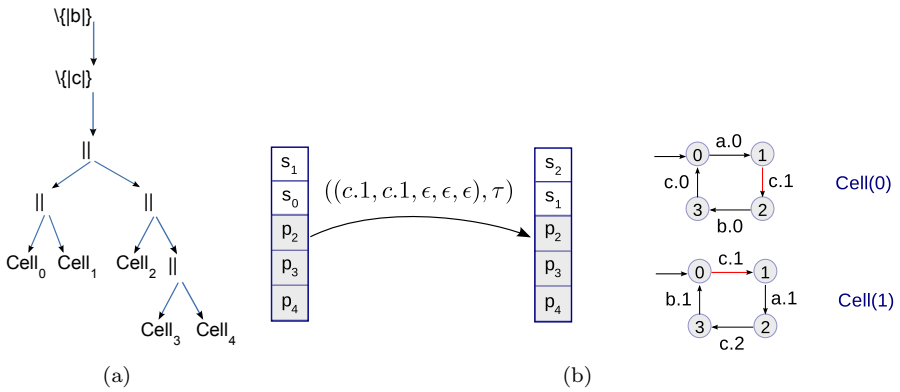

Figure 3: Operational semantics — Milner's scheduler

their LTS's generated explicitly. The root process contains just a single format with three types of supercombinators:

- if $Cell(i)$ and $Cell(i \oplus 1)$ perform $c.i \oplus 1$, $Scheduler \setminus \{|\, b \,|\}$ performs a τ
- if $Cell(i)$ performs $a.i$, $Scheduler \setminus \{|\, b \,|\}$ also performs $a.i$
- if $Cell(i)$ performs $b.i$, $Scheduler \setminus \{|\, b \,|\}$ performs a τ

Supercombinator operational representation can be considered an implicit LTS because it gives an initial state and sufficient information to calculate all the transitions of the system on-the-fly. Given a root high-level process, we refer to tuples of the current states of its leaf processes as *configurations*. 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 currently able to communicate the event that they are responsible for within the supercombinator. Hence, the operational semantics of the root process 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. For example, the supercombinator $((c.1, c.1, \epsilon, \epsilon, \epsilon), \tau)$ (see Figure 3(b)) would be enabled iff $Cell(0)$ is in its state s_1 and $Cell(1)$ is in its state s_0, independent of the current states of the other three cells. If this rule is enabled and the transition taken, $Cell(0)$ will go to state s_2, $Cell(1)$ will go to state s_1, the other three cells will not progress and $Scheduler \setminus \{|\, b \,|\}$ will perform a τ.

To summarise, supercombinators can be viewed as implicit state-space representations. They are generated by mimicking the SOS or combinator rules, but yield more compact storage and more efficient algorithms. Therefore, FDR is most efficient when manipulating processes with relatively simple sequential leaves composed in parallel or applied hiding or renaming upon. Of course, high-level processes can be explicated, i.e. transformed into explicit LTS's, paying a potentially exponential price. This is quite logical as explication breaks down the hierarchical structure of a system composed of concurrent processes and makes it sequential.

2.1.6. 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 $\mathsf{traces}(Impl) \subseteq$

$\mathsf{traces}(Spec)$.

FDR carries out the refinement check on the level of the LTS representations $OS_{Spec} = \langle S^s, s_0^s, L^s, T^s \rangle$ and $OS_{Impl} = \langle S^i, s_0^i, L^i, T^i \rangle$. The algorithm is similar to the standard one for deciding language containment $L(\mathcal{A}) \subseteq L(\mathcal{B})$ of nondeterministic automata $\mathcal{A}$ and $\mathcal{B}$, which reduces to checking whether $L(\mathcal{A}) \cap \overline{L(\mathcal{B})} = \emptyset$ and requires that $\mathcal{B}$ be determinised a priori. In a similar fashion, 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 bisimulation-reduced LTS. We remark that any finite-state CSP process can be normalised, although potentially incurring an exponential blow-up. 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 $\mathsf{initials}(s^i) \subseteq \mathsf{initials}(s^s)$. If the property is violated, the breadth-first mode of search guarantees that the counterexample generated is of minimal length.

2.2. SAT-Based Model Checking Techniques

In this section, we give a brief summary of SAT-based bounded model checking [BCCZ99] and temporal k-induction [ES03b].

2.2.1. Bounded Model Checking

Bounded model checking is a sound but generally incomplete technique that focuses on searching for counterexamples of bounded length only. The underlying idea is to fix a bound k and unwind the implementation 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 one of the following happens: (1) a counterexample is detected, (2) k reaches a precomputed threshold called *completeness threshold* [CKOS04; CKOS05], which indicates that the model satisfies the specification, or (3) the model checking instance becomes intractable.

Different notions of completeness threshold exist, mainly based on the properties of the underlying graph of the system, e.g. diameter (the longest shortest path between any two states), recurrence diameter (the longest simple path between any two states), forward and backward radius versions of both, the size of the state space, etc. [BCCZ99; CKOS04; CKOS05; BHvMW09]. A simple path is a path along which all states are different and, in general, the recurrence diameter of a graph can be arbitrarily longer than its diameter (the same holding for radii) — if we consider a clique of size n, it's diameter would be 1, while the recurrence diameter would be $n-1$. We remark that this problem is exacerbated when modelling concurrent systems due to the exponential blow up of the state space.

It is important to note that without knowing or reaching a completeness threshold, the BMC procedure is incomplete since we do not know at what step it is correct to stop iterating and declare that the system preserves the desired property. Therefore, BMC is mostly suitable for detecting bugs rather than for full verification, i.e. proving the absence of bugs.

The problem with completeness thresholds is two-fold. On one hand, calculating the exact completeness threshold can be as hard as the model checking problem itself [CKOS05] and, therefore, sound overapproximations of it are usually used in practice. On the other hand, in some cases those overapproximations can be too large to handle efficiently.

SAT-based BMC [BCCZ99] reduces the model checking problem to a propositional satisfiability problem. The idea is to construct at each step k 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 fast detection of counterexamples. Moreover, due to the iterative nature of the BMC framework, the counterexample generated is of minimal length.

In the original syntactic [BCCZ99] and the subsequent semantic [CKOS04; CKOS05] translation of BMC to SAT, the implementation is modelled by a Kripke structure M and verified against a specification f defined as an LTL formula. The BMC instance at each step k is translated to a Boolean formula $\varphi_k = [\![M]\!]_k \wedge [\![\neg f]\!]_k$, where $[\![M]\!]_k$ encodes all paths of M of length k and $[\![\neg f]\!]_k$ represents all paths of length up to k that violate f.

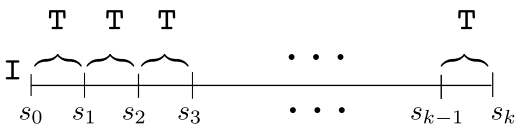

Figure 4: Encoding paths of length k

Generally, having a Boolean encoding of the state space (e.g. a binary or a one-hot encoding [KB05]), the Kripke structure M can be represented symbolically by a pair of Boolean functions $\langle I(s), T(s, s')\rangle$ defined as the characteristic functions of the set of initial states and the transition relation, respectively. We use s and s' as a shorthand for the vectors of Boolean variables necessary for encoding states of M. We replicate a separate copy of state variables s_i for each time step i. Then $[\![M]\!]_k = I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1})$ (see Figure 4). We illustrate the structure of the entire formula φ_k with a simple example in case $f = \mathsf{G}p$, where p represents a state predicate with Boolean encoding P:

$$\varphi_k = I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \wedge \bigvee_{i=0}^{k} \neg P(s_i)$$

2.2.2. Temporal k-Induction

Temporal k-induction [SSS00; ES03b] is a complete SAT-based technique for verifying safety properties. As opposed to BMC, it can be used also for establishing correctness of systems. Given a model $\langle I(s), T(s, s')\rangle$ and a safety property $P(s)$, the method checks if all reachable states of the model preserve P. k-induction builds upon BMC and is also conducted iteratively, as presented in Algorithm 1. It provides two conditions for termination in case the property is not violated — k-inductiveness of P for some $k \in \mathbb{N}$ or reaching the backward recurrence radius of the model with respect to P. The property P is k-inductive if it can be proven that if P holds along all paths of the system of length k, then it cannot be violated on a path of length $k + 1$. The backward recurrence radius is the length of the longest simple path from any state to a state violating P and is a valid completeness threshold.

For each step k, the temporal induction proof consists of two parts — a base case and an induction step. The base case $Base_k$ is similar to a BMC instance — we check if, starting from an initial state, there is a path of length k that violates P (see Figure 5). In the base case, we assume that we have already checked all base cases of shorter length and strengthen the BMC instance by stating that P holds along all initial paths of length up to $k - 1$. If the base case is satisfiable, we have found a counterexample. Otherwise, we proceed with the induction step $Step_k$ which is designed to prove that P is k-inductive. The induction step is strengthened and made complete by a constraint $Simple_k$

Algorithm 1 Temporal k-induction [ES03b]

1: **for** $k = 0$ to ∞ **do**
2: **if** *satisfiable*($Base_k$) **then**
3: return PROPERTY VIOLATED and a counterexample trace
4: **end if**
5: **if** *unsatisfiable*($Step_k \wedge Simple_k$) **then**
6: return PROPERTY HOLDS
7: **end if**
8: **end for**

requiring that all states on the $(k+1)$-path be different. Hence, k-induction terminates with a positive answer when reaching the backward recurrence r radius even if the property P has not manifested itself as k-inductive for any $k \leq r$.

$$\begin{array}{lcl} Base_k & \widehat{=} & I(s_0) \wedge \left(\bigwedge_{0 \leq i < k} (P(s_i) \wedge T(s_i, s_{i+1})) \right) \wedge \neg P(s_k) \\ Step_k & \widehat{=} & \left(\bigwedge_{0 \leq i < k+1} (P(s_i) \wedge T(s_i, s_{i+1})) \right) \wedge \neg P(s_{k+1}) \\ Simple_k & \widehat{=} & \bigwedge_{0 \leq i < j \leq k} (s_i \neq s_j) \end{array}$$

Figure 5: k-induction ingredients [ES03b]

We remark again that, in many cases, the backward radius of the model — the longest shortest path from any state to a state violating P, can be considerably smaller than its backward recurrence radius. However, the translation of shortest paths between two states to SAT involves plenty of existential quantifiers and is mostly suitable to using a QBF engine instead of a SAT solver.

As we are dealing with safety properties, we can also carry out the k-induction algorithm backwards, starting from states that violate P and trying to prove that initial states are never reachable. This can be implemented by redefining $Base_k$ and $Step_k$ as depicted in Figure 6. This algorithm guarantees termination upon reaching the forward recurrence radius — the longest simple path to any state starting from an initial state.

$$\begin{array}{lcl} Base_k & \widehat{=} & \neg P(s_0) \wedge \left(\bigwedge_{0 \leq i < k} (\neg I(s_i) \wedge T^{-1}(s_i, s_{i+1})) \right) \wedge I(s_k) \\ Step_k & \widehat{=} & \left(\bigwedge_{0 \leq i < k+1} (\neg I(s_i) \wedge T^{-1}(s_i, s_{i+1})) \right) \wedge I(s_{k+1}) \\ Simple_k & \widehat{=} & \bigwedge_{0 \leq i < j \leq k} (s_i \neq s_j) \end{array}$$

Figure 6: Backward k-induction ingredients

3. Bounded Trace Refinement Framework

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. Similarly to BMC and k-induction, we carry out the analysis on the level of the

operational representation of *Spec* and *Impl*. We point out that executions of length k can correspond to traces of smaller length if having τ actions entangled within, as defined in Section 2.1.4.

3.1. Challenges

As the LTS's underlying the operational semantics of processes are event-based models, we need to also handle events in our encoding. Let $OS_{Spec} = \langle I^s(s), T^s(s, l, s')\rangle$ and $OS_{Impl} = \langle I^i(t), T^i(t, l, t')\rangle$ be the models of *Spec* and *Impl*, respectively. At first glance, the most natural approach for encoding bounded execution refinement would be to try to directly mirror the original translation of BMC to SAT. We would need to similarly construct the Boolean formula φ_k as a conjunction of two formulae to model all executions of *Impl* of length k that are not executions of *Spec* — $\varphi_k = [\![OS_{Impl}]\!]_k \wedge [\![\neg OS_{Spec}]\!]_k$. Hence, we would be looking for an instantiation of the vectors of Boolean variables $l_1, \ldots, l_k$, such that $[\![OS_{Impl}]\!]_k = I^i(t_0) \wedge \bigwedge_{i=0}^{k-1} T^i(t_i, l_i, t_{i+1})$ is satisfiable and $[\![OS_{Spec}]\!]_k = I^s(s_0) \wedge \bigwedge_{i=0}^{k-1} T^s(s_i, l_i, s_{i+1})$ is not. Due to the implicit universal quantification of $s_0, \ldots, s_k$ in the unsatisfiability check of $[\![OS_{Spec}]\!]_k$, this analysis is mostly suitable to a QBF engine. Using a SAT-solver in this case would mean that we would need to extinguish all possible satisfying assignment of $l_1, \ldots, l_k$ in $[\![OS_{Impl}]\!]_k$ and to prove the unsatisfiability of $[\![OS_{Spec}]\!]_k$ over each one of them.

Furthermore, invisible τ actions can be arbitrarily interleaved in the executions of *Impl* and, therefore, syntactically different executions can produce semantically equivalent traces. This can lead to reporting spurious counterexamples. To illustrate this, consider the executions $\langle a, \tau, \tau, b, \tau, c\rangle$ of *Impl* and $\langle a, b, c\rangle$ of *Spec* as depicted on Figure 7(b). Even though they correspond to the same trace $\langle a, b, c\rangle$, they do not match pointwise and $\langle a, \tau, \tau, b, \tau, c\rangle$ would be falsely reported as a violation of *Spec*. However, bookkeeping the possible sequences of τ-s stuttered in between visible events does not seem to be trivial and computationally justifiable on the level of Boolean functions.

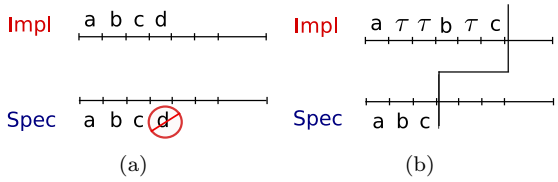

Figure 7: τ actions in bounded trace refinement

3.2. The Watchdog Approach

As explained in Section 2.1.6, FDR performs the refinement check by normalising the specification and looking for the existence of behaviours that the implementation allows and the specification does not.

As an alternative, the watchdog approach [RGM+03; Ros10] reduces the refinement check to analysing a single process constructed by composing the implementation in parallel with a transformed specification process. The latter plays the role of a watchdog that monitors the implementation and flags any behaviours that are considered violating with respect to the specification.

In our settings, using watchdog transformations allows us to actually reduce bounded execution containment to bounded reachability which is already amenable to SAT. The watchdog transformation phase is performed by means of FDR.

3.3. 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 hybrid two-level operational representation of processes, consisting of a process tree and a set of supercombinators, as defined in Section 2.1.5.

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

Without loss of generality, we assume that the implementation *Impl* comprises the interaction of c sequential processes $P_1, \ldots, P_c$ running in parallel, possibly using hiding, renaming or other CSP operators other than recursion. We write $Impl = P_1 \parallel P_2 \parallel \ldots \parallel P_c$ to actually denote a high-level process *Impl* with leaf processes $P_1, \ldots, P_c$, as defined in Section 2.1.5. This form of representing concurrent systems after supercompilation is of no limitation and, as mentioned in Section 2.1.1, we can handle the entire CSP_M syntax and functionality apart from the function *chase*. We use FDR to supercompile *Impl* and to retrieve both the set of supercombinators and the set $\{OS_{P_i} \mid i \in \{1, \ldots, c\}\}$.

3.4. Watchdog Bounded 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* and, in fact, many others, but marks those that do not conform to *Spec*. The transformation is carried out on the level of the LTS and not on the higher CSP description of *Spec*. It is most easily defined if the specification process is normalised so that it reaches a unique state after following any trace. The LTS of *Watchdog* is then obtained as an extension of OS_{Spec} — we add a fresh state *sink* and make OS_{Spec} total with respect to the alphabet $\alpha_{Spec} \cup \alpha_{Impl}$ by directing all non-existing transitions to *sink*. Formally, having $OS_{Spec} = \langle S^s, s_0^s, T^s, L^s = \alpha_{Spec} \rangle$, $OS_{Watchdog} = \langle S^w = S^s \cup \{sink\}, s_0^s, T^w, L^w = \alpha_{Spec} \cup \alpha_{Impl} \rangle$, where the transition relation T^w is defined as follows:

$$\begin{aligned} T^w \quad = \quad & T^s \cup \{(sink, l, sink) \mid l \in \alpha_{Spec} \cup \alpha_{Impl}\} \cup \\ & \{(s, l, sink) \mid s \in S^s, l \in \alpha_{Spec} \cup \alpha_{Impl}, s \stackrel{l}{\not\rightarrow}\} \end{aligned}$$

   The resulting process *Watchdog* operationally passes through *sink* whenever executing a trace that is not allowed by *Spec*.
2. We construct a process $Refinement = Watchdog \underset{\alpha_{Impl} \cup \alpha_{Spec}}{\parallel} Impl =$ $Watchdog \underset{\alpha_{Impl} \cup \alpha_{Spec}}{\parallel} (P_1 \parallel P_2 \ldots \parallel P_c)$. *Refinement* captures precisely the behaviours of *Impl*, but those behaviours that do not conform to *Spec* force *Watchdog* to bark, i.e. pass through its *sink* state. Hence, *Refinement* 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:

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

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

4. Boolean Encoding of CSP Processes

In this section we present our encoding of CSP processes into Boolean formulae. First, we show 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 illustrating the Boolean encoding in this section, we use the following notation. $\lceil X \rceil(\mathit{Vars})$ denotes the Boolean encoding of X with respect to the vector(s) of Boolean variables *Vars*.

Generally, to encode a finite set of elements S, we use an injective mapping $enc_S : S \rightarrow \{0,1\}^m$ to associate each element $s \in S$ with a unique bit vector $\overline{b} = (b_1, \ldots, b_m)$ of certain size m. To represent elements of S as Boolean functions, we introduce an ordered vector of m distinct Boolean variables $\overline{x} = (x_1, ..., x_m)$. Each variable x_i uniquely identifies the corresponding bit b_i and for each $s \in S$, $\lceil s \rceil(\overline{x})|_{\overline{x}=\overline{b}} = 1$ iff $enc_S(s) = \overline{b}$. Typically, binary or one-hot [KB05] encoding of sets are used in practice. The basic idea behind binary encoding is to enumerate the elements of S in binary notation and represent them as Boolean functions over $m = \lceil \log_2 |S| \rceil$ Boolean variables. In one-hot encoding, each $s \in S$ is represented by a bit vector of size $|S|$ in which precisely one bit is set to 1. To illustrate the two encodings, let us consider the set $S = \{s_0, s_1, s_2, s_3\}$. If we consider the binary encoding $s_0 = (00), s_1 = (01), s_2 = (10), s_3 = (11)$, a vector of just two variables $\overline{x} = (x_0, x_1)$ suffices and $\lceil s_1 \rceil(\overline{x}) = \neg x_0 \wedge x_1$, for example. For the one-hot encoding $s_0 = (1000), s_1 = (0100), s_2 = (0010), s_3 = (0001)$, we need a vector $\overline{x} = (x_0, x_1, x_2, x_3)$ of four variables and $\lceil s_1 \rceil(\overline{x}) = \neg x_0 \wedge x_1 \wedge \neg x_2 \wedge \neg x_3$. Alternatively, we can use $\lceil s_1 \rceil(\overline{x}) = x_1$, but in this case we need to add global constraints enforcing that precisely one bit is set to true at a given time instance. Those constraints can be expressed by a formula of size linear in $|S|$.

4.1. Encoding a Sequential Process

As explained in Section 3.3, for each sequential leaf process P, we obtain the explicit operational representation of P using FDR. Let $OS_P = \langle S, s_0, L = \Sigma^{\tau,\checkmark}, T \rangle$ be the LTS associated with the finite-state leaf process P communicating over a finite alphabet of events Σ. Using either binary or one-hot encoding of sets, we introduce vectors of Boolean variables $\overline{x}$ and $\overline{y}$ for encoding the set of states S and the set of labels L, respectively. We define $\lceil I \rceil(\overline{x}) = \lceil s_0 \rceil(\overline{x})$.

In order to represent the transition relation T, we employ a copy $\overline{x'}$ of $\overline{x}$. $\overline{x}$ serves to represent the source states of transitions and $\overline{x'}$ — 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: $\lceil T \rceil(\overline{x}, \overline{y}, \overline{x'}) = \bigvee_{t \in T} \lceil t \rceil(\overline{x}, \overline{y}, \overline{x'})$.

We can now represent a sequential process P implicitly by a pair of Boolean 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 $\mathit{Paths}(P, k)$ 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. In order to represent $\mathit{Paths}(P, k)$ symbolically, we replicate $(k+1)$ vectors of Boolean variables $\overline{x_0}, \overline{x_1} \ldots \overline{x_k}$ for encoding the states $s_0, s_1, \ldots, s_k$ and k vectors of Boolean variables

$\overline{y_1}, \overline{y_2} \ldots \overline{y_k}$ for the corresponding transitions $l_1, \ldots, l_k$. 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

In the setting of FDR, after supercompilation we can view a concurrent system as a high-level process identified by a process tree and a set of supercombinators. 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. In what follows, we denote by $Sys(c) = \langle\langle P_1, \ldots, P_c\rangle, SC\rangle$ the high-level process characterised by a set of supercombinator rules SC and c explicitly compiled leaf processes $P_1, ..., P_c$ communicating over sets of events $\Sigma_1, ..., \Sigma_c$, respectively. We define $\Sigma = \cup_{i=1}^{c} \Sigma_i$.

Encoding the Sequential Leaf Processes. For each $i \in \{1, ..., c\}$, we retrieve the explicit LTS representation $OS^i = \langle S^i, s_0^i, L^i = \Sigma_i^{\tau,\checkmark}, T^i\rangle$ of the leaf P_i from FDR. Since $\Sigma_i \subseteq \Sigma$, we actually consider $L^i = \Sigma^{\tau,\checkmark}$.

Following the ideas from the previous Section 4.1, we introduce vectors of Boolean variables $\overline{x^i}$, $\overline{x^{i'}}$ and $\overline{y^i}$ to generate the symbolic representation $\langle \lceil T^i \rceil(\overline{x^i}, \overline{y^i}, \overline{x^{i'}}), \lceil I^i \rceil(\overline{x^i})\rangle$ of P_i. Hence, each process has its own set of variables for representing the alphabet $\Sigma^{\tau,\checkmark}$. We further introduce an additional vector of Boolean variables $\overline{y}$ for encoding the resulting action of the entire system because, due to the presence of hiding and renaming, it might be different from the contributions of the leaf processes, as illustrated in Section 2.1.5. In case the system violates the specification, we also use the assignment of the variables from $\overline{y}$ to generate a counterexample trace.

Encoding Configurations of the Concurrent System. Recall that at, every time instance, the state of the entire high-level system, also called a *configuration*, is identified by the current states of its sequential leaf 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 2.1.5, supercombinators are rules for combining together actions of the individual sequential leaf 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 $\Sigma^\epsilon = \Sigma \cup \{\epsilon\}$. In addition, the set of supercombinators is partitioned into existing *formats*, i.e., different configurations of switched on and switched off processes among $P_1, \ldots, P_c$, which we denote by *Formats*.

Formally, the set of supercombinators can be represented as a $(c+3)$-ary relation $SC \subseteq Formats \times \Sigma_1^{\tau,\checkmark,\epsilon} \times ... \times \Sigma_c^{\tau,\checkmark,\epsilon} \times \Sigma^{\tau,\checkmark} \times Formats$, or more generally $SC \subseteq Formats \times (\Sigma^{\tau,\checkmark,\epsilon})^c \times \Sigma^{\tau,\checkmark} \times Formats$. $(f_{src}, a_1, ..., a_c, a, f_{dest}) \in SC$ 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* × *Configurations*
- set of labels – *SC*

- transition relation – $T \subseteq (\mathit{Formats} \times \mathit{Configurations}) \times SC \times (\mathit{Formats} \times \mathit{Configurations})$. 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
  $(f_i, a_1, \ldots, a_c, a, f_j) \in SC \wedge \forall_{k=1}^{c}((a_k \neq \epsilon \Rightarrow (s_i^k, a_k, s_j^k) \in T^k) \wedge$
  $(a_k == \epsilon \Rightarrow s_i^k = s_j^k))$.

Encoding Supercombinators. For a given rule $sc = (f_{src}, a_1, ..., a_c, a, f_{dest}) \in SC$, let $\mathit{Passive}(sc) = \{i \in \{1, \cdots, c\} \mid a_i = \epsilon$, 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:

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

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 Boolean 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, $\lceil sc \rceil(\overline{y^1}, ..., \overline{y^c}, \overline{y}, \overline{u}, \overline{f}, \overline{f'}) = \bigwedge_{i \notin \mathit{Passive}(sc)}(\lceil a_i \rceil(\overline{y^i}) \wedge \neg u_i) \wedge \bigwedge_{i \in \mathit{Passive}(sc)} u_i$
$\wedge \lceil a \rceil(\overline{y}) \wedge \lceil f_{src} \rceil(\overline{f}) \wedge \lceil f_{dest} \rceil(\overline{f'})$.

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 OS^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:

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

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:

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

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

5. Implementation Details

Our prototype tool SymFDR is written in C++ and uses FDR as a shared library for manipulating CSP processes. 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 or can be easily rewritten in this form.

5.1. BMC

In our BMC framework, we have three modes of state space traversal — forward (starting from the initial state), backward (starting from an error state) and simultaneous forward/backward mode.

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 so, 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 formulae with comparable size and structure. Hence, we trade off the shortness of the reported counterexample for efficiency.

5.2. k-induction

We have implemented the "Zig-Zag" and "Dual" temporal k-induction algorithms [ES03b]. The difference is that the "Dual" algorithm makes use of separate SAT-solvers for the base case and the induction step, aiming to optimise the incremental SAT interface. For k-induction, SymFDR supports both forward and backward traversal, yielding four algorithms in total: "Zig-Zag" forward, "Zig-Zag" backward, "Dual" forward and "Dual" backward.

5.3. SAT

SymFDR supports both binary and one-hot encoding of state spaces, though we have observed that for our test cases binary encoding scales much better. One-hot encoding usually yields CNF instances with smaller number of clauses but substantially higher number of Boolean variables which seems to burden the SAT solver. We construct the Boolean formulae directly in negation normal form and, consequently, transform them into equisatisfiable formulae in CNF using the optimised one-sided Tseitin encoding [BKWW08; BHvMW09].

Currently, SymFDR supports MiniSAT 2.0, PicoSAT 846 and ZChaff, all working in incremental mode. For our test cases, we have found MiniSAT to be the most efficient and all quoted results use MiniSAT. We exploit MiniSAT's incremental interface in a way similar to [ES03b]. 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 (learntsize_factor = 0.2, learntsize_inc = 1.02) and restart more frequently (restart_inc = 1.1). We also implemented adding unit learned clauses explicitly, as suggested in [ES03a], in conjunctions of multiple ones. Using positive polarity in decision heuristics also produced much better results, as well as freezing and then defreezing state and format variables at each step to avoid variable elimination.

SymFDR also supports strategies for 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 2.83, FDR used in a non-standard way, PAT 3.2.2 [SLD08], and, in some cases, direct SAT encodings, NuSMV 2 [CCG+02] and Alloy Analyzer 4.1.10 [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 GB RAM running Linux, except the test marked with a *, which was performed on a 4-GB-RAM PC running Linux.

6.1. Tools We Compare Against

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, partial-order reductions, counter abstraction, symmetry reduction, etc. In the test cases quoted here, the specification is given as a reachability property on the values of the shared variables, as modelled in the benchmarks available with the tool. 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.

6.2. Test Cases

6.2.1. Instances with Counterexamples

In this section, we consider test cases with counterexamples and therefore exploit the BMC framework. The results are summarised in Table 1, Table 2, Table 3, Table 4, Table 5 and Table 6. The last column titled ♯ lists the length of counterexamples.

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. The performance of SymFDR and PAT seems to alternate. The explicit DFS-based tools FDR-Div and PAT seem to perform quite unevenly, signifying the importance of luck. We have also observed that in cases where a counterexample does not exist, FDR's BFS strategy outperforms the DFS-based tools PAT and SymFDR. To give an idea of the size of the SAT instances that SymFDR generates, for $N = 32$, the instance originally contains 314 567 clauses using 38 034 variables. MiniSAT learns extra 132 451 clauses and finds a satisfying assignment after 132 056 844 propagations. In comparison, FDR finds a solution after traversing 187 000 000 states.

N	FDR	Time (sec.)				SAT	♯
	♯ states checked	FDR	FDR -Div	PAT	SymFDR	freq.	
20	41 703	0	0	5.17	5.75	10	20
					18.14	20	
23	411 976	5	0	1.83	13.1	12	23
					7.69	23	
26	4 048 216	72	0	6.23	22.81	13	26
					21.47	26	
29	28 249 254	581	1	70.64	34.35	15	29
					17.35	29	
32	$>$ 139 000 000	$>$ 11 700	5	7.08	147.68	16	32
	187 000 000*	2 640*			66.3	32	
35	—	—	1 485	484.25	214.63	18	35
					90.97	35	
38	—	—	43	3 786.18	182.91	19	38
41	—	—	4	358.69	325.59	41	41

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

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 $N = 6$, SymFDR found a solution in approximately 5 minutes, while FDR crashed after an hour and a half of state-space exploration. For this test case, we have observed that restricting the decision variables in MiniSAT to the input ones enhances the performance of SymFDR, especially for $N = 7$ where the reduction factor is over 20. Hence, for $N = 7$, the performance of the general tool SymFDR comes close to the performance of the problem-specific SAT encoder for the chess knight tour.

We have observed similar results with the test cases of finding a Hamiltonian path on an $N \times M$ grid (Table 3) and the lights off puzzle (Table 4). The lights off puzzle starts with an $N \times N$ board with all lights initially on. The aim is to reach a configuration with all lights switched off having in mind that upon triggering any light switch, the switch to the right, left, below and above is also triggered. This test case illustrates the difference that search mode can make in SymFDR. We remark that for every N, if using inductive normal compression, FDR can actually obtain a state space consisting of a single state which it can verify in 0 seconds.

N	FDR	Time (sec.)					SAT	♯
	♯ states checked	FDR	FDR -Div	PAT	Direct SAT	SymFDR	freq.	
5	508 451	3	0.147	0.53	8.5	8.15	13	26
6	> 120 000 000	—	18	17.17	125.3	298.64	19	37
7	—	—	—	12.86	1 138.0	1 326.18	25	50

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

$N \times M$	FDR	Time (sec.)			SAT	♯
	♯ states checked	FDR	FDR -Div	SymFDR	freq.	
4×4	2 518	0	0	0.83	9	17
4×5	14 368	0	0	1.98	11	21
5×6	1 219 416	21	33	32.9	16	31
				19.27	31	
6×6	18 115 326	243	630	43.54	19	37
6×7	> 115 000 000	> 3 600	> 3 600	228.88	22	43

Table 3: Performance comparison – Hamiltonian cycle on a $N \times M$ grid ($\sharp = N \times M + 1$)

N	FDR	Time (sec.)					SAT	♯
	♯ states checked	FDR	PAT	SymFDR fw	SymFDR bw	SymFDR fw/bw	freq.	
2	16	0		0.05	0.05	0.06	5	5
3	382	0		0.16	0.14	0.19	6	6
				0.23	0.24	0.32	5	
4	2 084	0		0.26	0.27	0.33	5	5
5	8 388 608	537	> 7 600	—	23.45	655.33	8	16
6	—	out of mem		—	—	—	15	28

Table 4: Performance comparison – lights off puzzle on a $N \times N$ board (♯ unpredictable)

The fifth 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 5. NuSMV-BMC and SymFDR seem to be competitive, both outperforming Alloy Analyzer. SymFDR working in simultaneous forward/backward mode outperforms NuSMV-BMC. However, all non-SAT tools — the explicit ones FDR and PAT and the BDD-based NuSMV — are clearly orders of magnitude more efficient than the SAT-based ones. We remark, though, that all solutions for the puzzle generated by PAT are longer than 1000 moves, even when $N = 5$, when the shortest solution is of length 32. When configuring PAT to report the shortest witness trace, we obtain the results quoted in the column labelled "PAT short". In this case, the performance of PAT worsens fast, falling behind SymFDR for $N = 7$ and running out of memory for $N = 8$.

N	Time (sec.)								SAT	♯
	FDR	PAT	PAT short	Nu SMV	SymFDR fw	SymFDR fw/bw	Alloy	NuSMV -BMC	freq.	
5	0.19	0.21	0.98	0.43	4.9	3.6	11	2.2	16	32
6	0.20	1.18	10.15	0.66	27.3	21.6	327	34.9	32	64
7	0.16	2.18	202.4	0.17	182.7	173.3	21 537	1 865	64	128
8	0.18	15.20	—	0.29	3 114.1	2 035.1	—	2 218	128	256

Table 5: Performance comparison – Hanoi towers with N disks ($\sharp = 2^N$)

Our final Table 6 summarises results obtained while running SymFDR on CSP scripts generated by *Casper* [Low98] — a well-known tool for analysing and verifying the correctness of security protocols, underlying the discovery of an attack on the Needham Schroeder public key protocol in 1995 [Low95] and the verification of correctness of a fixed version of it in 1996 [Low96]. Casper takes a big advantage of the partial-order-reduction function *chase* offered by FDR [Ros98], as is apparent from the comparison of the performance of FDR with and without it. For the Needham Schroeder public key protocol (NSPK3), SymFDR is better than FDR without *chase* but worse than FDR with *chase*. SymFDR finds those instances particularly hard because, on one hand, the state-space blow-up is enormous and, on the other hand, probably due to the great number of τ actions entangled into the state space, there are very few clauses learned and those clauses contain far too many literals (often over 1000). For these test cases, we have used MiniSAT configured with negative polarity and no decision on auxiliary variables.

Protocol	Time (sec.)					SAT	♯
	FDR *chase*	FDR no *chase*	SymFDR fw	SymFDR bw	SymFDR fw/bw	freq.	
NSPKP3	1	452	93.37	—	—	7	14
TMN1	0	0	34.82	36.89	30.73	7	7
Andrew	0	0	4.35	4.03	3.97	8	8

Table 6: Performance comparison – (violated) security protocols

6.2.2. Instances without Counterexamples

In this section we focus on the performance of SymFDR using k-induction (see Table 7). For each of the four algorithms, "Zig-Zag" forward, "Zig-Zag" backward, "Dual" forward and "Dual" backward, we record the time in seconds and the step at which the algorithm terminates.

We consider the readers/writers test case, Milner's scheduler with and without compression, the Bakery algorithm for mutual exclusion and the bully algorithm for leader election. Besides manually generated scripts, we have experimented with scripts translated by Casper and SVA (Shared Variables Analyser) [Ros10] — a front-end for FDR based on modelling concurrency using shared variables. We

Test	Zig-zag forward		Zig-zag backward		Dual forward		Dual backward	
	Time	k	Time	k	Time	k	Time	k
R/W 100	4.69	5	1.2	2	7.15	4	3.07	2
R/W 200	17.77	5	4.09	2	27.64	4	10.94	2
R/W 300	—	—	8.61	2	—	—	—	—
R/W 400	—	—	15.95	2	—	—	—	—
R/W 500	—	—	25.84	2	—	—	—	—
R/W 600	—	—	35.54	2	—	—	—	—
R/W 700	—	—	49.00	2	—	—	—	—
Milner 2	0.07	5	0.09	8	0.08	5	0.12	7
Milner 3	0.65	14	1.78	20	0.63	13	1.17	19
Milner 4	559.05	33	2181.07	50	376.03	32	57.86	49
Milner 2 leaf normal	0.03	3	0.04	4	0.02	3	0.05	4
Milner 3 leaf normal	0.17	9	0.08	6	0.21	8	0.13	6
Milner 4 leaf normal	0.67	13	0.19	8	0.64	12	0.27	8
Milner 5 leaf normal	6.17	20	0.38	10	6.55	19	0.48	10
Milner 6 leaf normal	306.84	29	0.75	12	225.15	28	0.88	10
Milner 10 leaf normal	—	—	4.84	20	—	—	7.13	20
Milner 15 leaf normal	—	—	27.58	30	—	—	21.64	30
Milner 20 leaf normal	—	—	155.1	40	—	—	103.53	40
Bakery 2 (4), SVA hc	0.14	2	0.26	9	0.15	2	—	—
Bakery 2 (8), SVA hc	0.29	2	1.86	21	0.29	2	—	—
Bakery 2 (16), SVA hc	0.89	2	17.4	45	0.83	2	—	—
Bakery 2 (32), SVA hc	2.92	2	—	—	2.98	2	—	—
Bakery 2 (64), SVA hc	12.9	2	—	—	13.53	2	—	—
Bakery 3 (4), SVA hc	—	> 33	—	—	—	—	—	—
Bully 3 (1,2,3)	—	—	152.95	40	—	—	—	—
Bully 3 (1,3,7)	—	—	151.97	40	—	—	—	—

Table 7: Performance comparison – k-induction

have also considered the effect of applying available FDR compression techniques to the CSP scripts and have analysed the impact of those techniques to the the recurrence radii.

In our experience, the backward algorithm, aiming to reach the forward recurrence radius, often scales better than the forward one. We note that, due to concurrency, the completeness threshold blows up in all cases. Hence, successful performance mainly depends on whether the property is inductive or not. For the Bakery algorithm and Milner's scheduler, applying hierarchical and leaf compression, respectively, has proven to be beneficial and has significantly decreased the termination step and, hence, improved the performance. For all four algorithms we have observed that the induction step is checked much faster than the base one, opposite to what reported in [ES03b; SSS00]. Hence, we also implemented versions of k-induction starting the iteration process from a step greater that zero, as suggested in [SSS00]. Opposite to what we expected, though, for our test cases this approach scales worse than the standard one. In case of unsatisfiable instances, we have also observed that if the length of the longest possible counterexample is known in advance, often iterating the BMC

algorithm up to this length produces better results than k-induction due to tuning the SAT frequency and jumping multiple time steps at once.

6.2.3. Conclusion

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.

SymFDR in k-induction mode works reasonably well for small test cases, especially if the property is inductive. However, for larger test cases, SymFDR does not scale very well as the completeness threshold becomes too large due to concurrency. In all cases considered in Table 7, FDR is considerably faster.

7. Conclusions and Future Work

In this paper we have demonstrated the feasibility of integrating SAT-based BMC and k-induction 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.

We are currently implementing 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 backward radius of the state-space which is smaller than its backward recurrence radius, as is the case with temporal induction. Moreover, experimental results have shown that, in practice, the algorithm often converges substantially faster, for bounds considerably smaller than the backward radius. 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 is the foundation we build upon.

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

Acknowledgments

We are grateful to the anonymous reviewers for their thorough and truly helpful feedback, to Daniel Kroening and James Worrell for their early comments, Philip Armstrong for his help with FDR and

Vijay D'Silva, Alastair Donaldson and Phillip Rümmer for interesting discussions on k-induction. The analysis using DFS refinement through divergence checking was inspired by a correspondence several years ago between A. W. Roscoe and James Heather. The work presented in this paper is supported by grants from EPSRC and US ONR.

References

[BCCZ99] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In *TACAS*, pages 193–207. Springer-Verlag, 1999.

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

[BHvMW09] Armin Biere, Marijn J. H. Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, February 2009.

[Bie08] Armin Biere. Picosat essentials. *JSAT*, 4(2-4):75–97, 2008.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press, 2008.

[BKWW08] Armin Biere, Daniel Kroening, Georg Weissenbacher, and Christoph Wintersteiger. *Digitaltechnik*. Springer, 2008.

[CCG+02] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. In *CAV*, volume 2404 of *LNCS*. Springer, July 2002.

[CCO+05] Sagar Chaki, Edmund M. Clarke, Joël Ouaknine, Natasha Sharygina, and Nishant Sinha. Concurrent software verification with states, events, and deadlocks. *Formal Aspects of Computing*, 17(4):461–483, 2005.

[CGJ+00] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *CAV '00: Proceedings of the 12th International Conference on Computer Aided Verification*. Springer LNCS, 2000.

[CGP99] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, 1999.

[CKOS04] Edmund M. Clarke, Daniel Kroening, Joël Ouaknine, and Ofer Strichman. Completeness and complexity of bounded model checking. In *VMCAI*, pages 85–96, 2004.

[CKOS05] Edmund M. Clarke, Daniel Kroening, Joël Ouaknine, and Ofer Strichman. Computational challenges in bounded model checking. *STTT*, 7(2):174–183, 2005.

[COYC03] Sagar Chaki, Joël Ouaknine, Karen Yorav, and Edmund M. Clarke. Automated compositional abstraction refinement for concurrent C programs: A two-level approach. In *Electronic Notes in Theoretical Compututer Science*, volume 89, 2003.

[EB05] Niklas Eén and Armin Biere. Effective preprocessing in SAT through variable and clause elimination. In *SAT*, pages 61–75, 2005.

[ES03a] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In *SAT*, pages 502–518, 2003.

[ES03b] Niklas Eén and Niklas Sörensson. Temporal induction by incremental SAT solving. In *Electronic Notes in Theoretical Compututer Science*, volume 89, 2003.

[FSE05] FSEL. *Failures-Divergence Refinement. FDR2 User Manual.* Formal Systems (Europe) Ltd., June 2005.

[Gol04] Michael Goldsmith. Operational semantics for fun and profit. In *25 Years Communicating Sequential Processes*, pages 265–274, 2004.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1985.

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

[KB05] Randy H. Katz and Gaetano Borriello. *Contemporary Logic Design, Second Edition.* Prentice Hall, 2005.

[Low95] Gavin Lowe. An attack on the Needham-Schroeder public-key authentication protocol. *Information Processing Letters*, 56(3):131–133, 1995.

[Low96] Gavin Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *TACAS '96: Second International Workshop on Tools and Algorithms for Construction and Analysis of Systems*, volume 1055 of *Lecture Notes in Computer Science*, pages 147–166. Springer, 1996.

[Low98] Gavin Lowe. Casper: A compiler for the analysis of security protocols. *Journal of Computer Security*, 6(1-2):53–84, 1998.

[McM03] Kenneth L. McMillan. Interpolation and SAT-based model checking. In *CAV*, pages 1–13, 2003.

[Pel98] Doron Peled. Ten years of partial order reduction. In *CAV '98: Proc. 10th International Conference on Computer Aided Verification*, pages 17–28, London, UK, 1998. Springer-Verlag.

[PY96] A. Parashkevov and J. Yantchev. ARC - a tool for efficient refinement and equivalence checking for CSP. In *AAPP*, 1996.

[RGM+03] A. W. Roscoe, M.H. Goldsmith, N. Moffat, T. Whitworth, and I. Zakiuddin. Watchdog transformations for property-oriented model checking. In *Proc. FME*, 2003.

[Ros88] A. W. Roscoe. Unbounded nondeterminism in CSP. Technical Report PRG-67, Oxford University Computing Laboratory, July 1988. In *Two papers on CSP.* Also appeared in *Journal of Logic and Computation*, Vol 3, No 2 pages 131-172 (1993).

[Ros94] A. W. Roscoe. Model-checking CSP. In A. W. Roscoe, editor, *A Classical Mind: essays in Honour of C.A.R. Hoare*, chapter 21. Prentice-Hall, 1994.

[Ros98] A. W. Roscoe. *The Theory and Practice of Concurrency.* Prentice Hall, 1998.

[Ros10] A. W. Roscoe. *Understanding Concurrent Systems.* Springer, 2010.

[Ros11] A. W. Roscoe. On the expressiveness of CSP. To appear. Available at http://www.cs.ox.ac.uk/files/1383/expressive.pdf, February 2011.

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

[Sht00] Ofer Shtrichman. Tuning SAT checkers for bounded model checking. In *CAV '00: Proceedings of the 12th International Conference on Computer Aided Verification*, pages 480–494, London, UK, 2000. Springer-Verlag.

[SLD08] Jun Sun, Yang Liu, and Jin Song Dong. Model checking CSP revisited: Introducing a process analysis toolkit. In *ISoLA*, pages 307–322, 2008.

[SLDS08] Jun Sun, Yang Liu, Jin Song Dong, and Jing Sun. Bounded model checking of compositional processes. In *TASE*, pages 23–30. IEEE, 2008.

[SSS00] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking safety properties using induction and a SAT-solver. In *FMCAD '00: Proceedings of the Third International Conference on Formal Methods in Computer-Aided Design*, pages 108–125, London, UK, 2000. Springer-Verlag.