How to Make FDR Spin

LTL Model Checking of CSP by Refinement

Michael Leuschel[1], Thierry Massart[2] and Andrew Currie[1]

[1] Department of Electronics and Computer Science
University of Southampton
Highfield, Southampton, SO17 1BJ, UK
e-mail: {ajc,mal}@ecs.soton.ac.uk

[2] Computer Science Department
University of Brussels ULB - CP 212
Bld du Triomphe
B-1050 Brussels, Belgium
e-mail: tmassart@ulb.ac.be

Abstract. We study the possibility of doing LTL model checking on CSP specifications in the context of refinement. We present evidence that the refinement-based approach to verification does not seem to be very well suited for verifying certain temporal properties. To remedy this problem, we show how to (and how not to) perform LTL model checking of CSP processes using refinement checking in general and the FDR tool in particular. We show how one can handle (potentially) deadlocking systems, discuss the validity of our approach for infinite state systems, and shed light on the relationship between "classical" model checking and refinement checking.

1 Introduction

Recent years have seen dramatic growth [8] in the application of model checking [7, 4] techniques to the validation and verification of correctness properties of hardware, and more recently software systems.

One of the methods is to model a hardware or software system as a finite, labeled transition system (LTS) which is then exhaustively explored to decide whether a given temporal logic specification ϕ holds, i.e., *checking* whether the system is a *model* of the formula ϕ. This approach has lead to various implementations, such as SPIN [14] for model checking of formulas in LTL [3] or SMV [17] for CTL [7] model checking.

Another, quite different, approach is based on the notion of *refinement* and is used by tools such as FDR [18, 11]. Here, the idea is to model *both* the system and the property in the *same* formalism, e.g., as CSP [13] processes. A system is said to satisfy a property ϕ if it is a refinement of ϕ. In CSP, refinement can be defined as language containment, failures containment, or failures and divergences containment.

The refinement-based approach suits itself very nicely to the stepwise development of systems, while the temporal logic approach often allows for more natural or succinct temporal specifications. It is quite surprising that the relationship between these two approaches appears not to have been studied. For instance, on the practical side, one might be interested in using tools such as FDR to do classical temporal logic model checking of CSP processes. In that context it would be interesting to know how (subclasses of) LTL or CTL temporal logic formulas can be translated into refinement tests. On the theoretical side, one might be interested in studying the expressive power of full LTL or CTL compared to refinement based model checking.

In this paper, we study the possibility of doing LTL model checking on CSP specifications in the context of refinement in general and using the FDR tool in particular. We discuss some unfruitful attempts at this translation, which show that is surprisingly difficult to find intuitive formulations of classical model checking tasks as refinement checks. In particular, this means that a tool such as FDR can currently not be used by ordinary users to perform LTL or CTL model checking. This is a pity, as FDR can handle full CSP extended with functions and advanced datatypes (such as lists, integers, sets), thus providing a powerful specification and prototyping language, and it would be extremely valuable to apply "classical" model checking to such specifications (e.g., to validate the initial specification). To remedy this problem, we then present a translation from LTL to refinement (based on Büchi automaton) which *does* work and (once automated) allows one to easily perform "classical" model checking in a CSP setting with refinement.

The remaining part of this paper contains the following. Sect. 2 contains the basic definitions of the subset of CSP we use and of LTL. Section 3 and 4 describes our (unsuccessful and successful) attempts at translating LTL model checks into refinement checks. In particular, we discuss the semantics of finite versus infinite traces and define LTL$^{\Delta}$ which can express properties both on infinite and deadlocking traces. We give our construction for a tester which allows us to achieve the model-checking using refinement. In Sect. 6 we end with a discussion of complexity and future work.

2 Preliminaries

Let us first briefly recall the main definitions for the CSP language, and for the LTL logic. A more complete and motivated definition can be found in [18] for CSP and in [6] for LTL.

CSP and Refinement CSP is a process algebra defined by Hoare [13]. The first semantics associated with CSP was a denotational semantics in terms of traces, failures and (failure and) divergences. An important notion is refinement: P refines Q denoted by $P \sqsupseteq Q$, iff $[\![P]\!] \subseteq [\![Q]\!]$, where $[\![P]\!]$ stands for the (particular) semantics of P. E.g., trace refinement is no more than language containment. Also, P is said to be *equivalent* to Q *iff* P refines Q and Q refines P. CSP also has an operational semantics defined, e.g., in [18].

Let us now give the syntax and semantics of the subset of CSP we want to handle. This subset will be sufficient to illustrate the problems as well as the possible solutions for doing LTL model checking using refinement.

Given Σ, a finite or enumerable set of actions (which we will henceforth denote by lower case letters $a, b, c, \ldots$), and $\mathcal{X}$, an enumerable set of variables or processes (which we henceforth denote by identifiers such as $Q, R, \ldots$, or $MYPROCESS$ starting with an uppercase letter), the syntax of a basic CSP expression is defined by the following grammar (where A denotes a set of actions):

P ::=

$STOP$ (deadlock) |	$a \rightarrow P$ (prefix) |
$P \sqcap P$ (internal choice) |	$P \,\square\, P$ (external choice) |
$P \,[\![A]\!]\, P$ (parallel composition) |	$P \backslash A$ (hiding) |
Q (instantiation of a process)	

Moreover, each process Q used must have a (possibly recursive) definition $Q = P$.

We suppose that all used processes are defined by at least one recursive definition (if there is more than one definition this is seen to be like an external choice of all the right-hand sides). In the following, we also suppose the alphabet Σ to be finite.

Intuitively, $a \to P$ means that the system proposes the action a to its environment, which can decide to fire it. The external choice is resolved by the environment (except when two branches propose the same action, where a nondeterministic choice is taken in case the environment chooses that action). Internal choice is made by the system without any control from the environment. $P \llbracket A \rrbracket Q$ is the generalized parallel operator of [18], and means that the process P synchronizes with Q on any action in the set of actions A. If an action outside A is enabled in P or Q, it can occur without synchronization of both processes. Pure interleaving $P \llbracket \emptyset \rrbracket Q$ is denoted by $P \mid\mid\mid Q$. Pure synchronization $P \llbracket \Sigma \rrbracket Q$ is denoted by $P \mid\mid Q$. The hiding operator $P \backslash A$ replaces any visible action a of A, P has by the internal action τ.

Note that the internal action τ is a particular action distinct from any action of Σ (called visible actions). Intuitively this internal action allows to denote a transition of the system from one state to another without any visible result from outside the system. In CSP, we handle *visible traces*, i.e. traces where τ actions have been removed.

In the trace semantics, the meaning $\llbracket P \rrbracket$ of a process P is the prefix closed set of all the visible *finite* traces of P. The failure semantics additionally assigns to a process P the set $\mathit{failures}(P)$ of couples: the first element is a visible finite trace t of the process P and the second component is a set R of refusals, i.e. the set of all sets of actions the process P can refuse after having performed the finite trace t. The divergence semantics of CSP also assigns to a process P the set $\mathit{divergences}(P)$ of traces after which the process can diverge, i.e., perform an infinite number of invisible actions τ in sequence.

We shall denote $P \sqsupseteq_{\mathcal{T}} Q$ and $P \sqsupseteq_{\mathcal{F}} Q$ if the process P is resp. a trace or a failure refinement of Q. Note that these semantics are slightly different from classical CSP. In classical CSP, an immediately diverging process is equivalent to $CHAOS$, which can perform any sequence of actions and refuse any set of actions at any point. In our context, a diverging process cannot perform all possible traces and failures, which conforms to the traces-refinement and failure-refinement implemented in the FDR tool. A formal definition of the various semantics of CSP can be found in [18, 13]

Example 1. Take $\Sigma = \{a, b\}$, $P_1 = a \to STOP$, and $P_2 = (a \to STOP) \sqcap (b \to STOP)$. Then $P_1 \sqsupseteq_{\mathcal{F}} P_2$ because $\mathit{failures}(P_1) = \{(\epsilon, \{b\}), (\epsilon, \emptyset), (a, \Sigma), (a, \{a\}), (a, \{b\}), (a, \emptyset)\}$ and $\mathit{failures}(P_2) = \{(\epsilon, \{b\}), (\epsilon, \emptyset), (\epsilon, \{a\}), (a, \Sigma), (a, \{a\}), (a, \{b\}), (a, \emptyset), (b, \Sigma), (b, \{a\}), (b, \{b\}), (b, \emptyset)\}$. Observe that $(\epsilon, \{a, b\}) \notin \mathit{failures}(P_2)$, i.e., P_2 cannot refuse both a and b (but it will refuse either a or b depending on the internal choice).
Also, $P_1 \not\sqsupseteq_{\mathcal{F}} (a \to STOP) \,\Box\, (b \to STOP)$ because $\mathit{failures}((a \to STOP) \,\Box\, (b \to STOP)) = \{(\epsilon, \emptyset), (a, \Sigma), (a, \{a\}), (a, \{b\}), (a, \emptyset), (b, \Sigma), (b, \{a\}), (b, \{b\}), (b, \emptyset)\}$.

LTL LTL [3] is a linear-time temporal logic, in the sense that it uses a trace semantics. Given an alphabet Π of elementary propositions (which we denote by lower-case letters such as $a, b, c, \ldots$), the syntax of LTL is given by the following grammar:

$$\phi ::= \mathit{false} | \mathit{true} | a | \neg a | \phi \wedge \phi | \phi \vee \phi | \bigcirc \phi | \phi \,\mathcal{U}\, \phi | \phi \,\mathcal{R}\, \phi$$

Note that LTL is usually defined for state based models (i.e., Kripke structures) while the operational semantics of CSP provides a labelled transition system where transitions and not states carry labels, some of the transitions being labelled by the invisible action τ. We thus have to be very careful about what the meaning of an elementary formula a is and what the concept of a successor state (in light of τ) is[1].

First, we will set Π to be identical to the set of actions Σ used within CSP processes. Second, as usual in LTL, we will define the meaning of formulas on individual traces of a system, and a system is a model for a formula iff all its traces satisfy the formula. This definition means that the LTL formula a is true for a system iff the system can perform a visible action (possibly after a sequence of invisible ones) and that in all cases this *visible action must* be a. Also, the system is a model for $\neg a$ iff the action a can not be fired as first visible action.

Conjunction and disjunction have the usual meaning. $\bigcirc$ is the next operator; e.g. $\bigcirc\phi$ means that the system can always perform a visible action, and that after this action, the formula ϕ must be true. Notice that various invisible actions may occur before the first visible action i.e., in our context this operator is not a "next state" operator but a "next after visible action" operator. $\phi\, \mathcal{U}\, \psi$ means that for every execution of the system the formula ψ must eventually become true and furthermore the formula ϕ must be true until (but not necessarily including) the first point at which ψ becomes true. $\mathcal{R}$ is the release operator which is the dual of the $\mathcal{U}$ operator; $\phi\, \mathcal{R}\, \psi$ intuitively means that ψ must be true up until and including the first point at which ϕ becomes true (but ϕ need not necessarily ever become true).

Formal semantics: The truth value of an LTL formula is first defined individually for each valid trace of the system (rather than on the whole labelled transition system). Usually, these traces are supposed to be infinite, i.e., deadlocking is not allowed. Later in the paper, we will remedy this limitation by extending the finite, deadlocking traces with an infinite number of special "Δ" actions.

First, given an infinite trace $\pi = \pi_0, \pi_1, \ldots$ we define π^i to be the trace $\pi_i, \pi_{i+1}, \ldots$. We now define $\pi \models \phi$ (a trace π satisfies or is a model of a formula ϕ) as follows:

- $\pi \not\models \mathit{false}$
- $\pi \models \mathit{true}$
- $\pi \models a$ iff $\pi_0 = a$
- $\pi \models \neg a$ iff $\pi_0 \neq a$
- $\pi \models \phi \wedge \psi$ iff $\pi \models \phi$ and $\pi \models \psi$
- $\pi \models \phi \vee \psi$ iff $\pi \models \phi$ or $\pi \models \psi$
- $\pi \models \bigcirc\phi$ iff $\pi^1 \models \phi$
- $\pi \models \phi\, \mathcal{U}\, \psi$ iff there exists a $k \geq 0$ such that $\pi^k \models \psi$ and $\pi^i \models \phi$ for all $0 \leq i < k$
- $\pi \models \phi\, \mathcal{R}\, \psi$ iff for all $k \geq 0$ such that $\pi^k \models \neg\psi$ there exists an $i, 0 \leq i < k$ such that $\pi^i \models \phi$

Moreover, two additional (derived) operators are usually defined: the always ($\Box$) and the *eventually* ($\Diamond$) operators: $\Diamond\phi \equiv \mathit{true}\, \mathcal{U}\, \phi$ and $\Box\phi \equiv \neg\Diamond\neg\phi$.

As is well known, any LTL formula $\neg\phi$ can be normalized into a form where negation is only applied to elementary propositions.

Finally, a non-deadlocking system S satisfies a formula ϕ, denoted by $S \models \phi$, if all its infinite traces satisfy ϕ: $S \models \phi$ iff $\forall \pi \in [\![S]\!]_\omega$, $\pi \models \phi$, where $[\![S]\!]_\omega$ is the set of the infinite traces of S. Note that in LTL $S \not\models \phi$ does *not* imply $S \models \neg\phi$ (although for each individual trace we have $\pi \not\models \phi$ iff $\pi \models \neg\phi$).

[1] We do not have to handle "tick" to mark the termination of a process as we do not treat SKIP or sequential composition.

3 Model Checking Using a Specification and Refinement

We report on our first attempts to do LTL model checking using the classical refinement based approach, i.e., writing a *specification* describing all admissible behaviours and then checking that our system is a valid *refinement* of that specification. As we will show below this turns out to be surprisingly difficult, or even impossible.

3.1 The obvious approach

Let us first try to solve the problem for systems S which do not deadlock. If we denote by $\llbracket\phi\rrbracket_\omega$ the set of infinite traces which satisfy the formula ϕ, we have $S \models \phi$ iff $\llbracket S\rrbracket_\omega \subseteq \llbracket\phi\rrbracket_\omega$. The link between LTL model checking and trace refinement is thus obvious and model checking corresponds to language containment. If we succeed in building a process $Spec_\phi$ which generates all the traces that satisfy ϕ, we could try to use trace refinement to do LTL model checking. Unfortunately, refinement in FDR and CSP[2] is based on *finite* traces only and a simple example suffices to show that a finite traces refinement test $S \sqsupseteq_{\mathcal{T}} Spec_\phi$ is, in general, not adequate to model check $S \models \phi$.

Example 2. Indeed, $S \sqsupseteq_{\mathcal{T}} Spec_\phi$ *iff* $\llbracket S\rrbracket \subseteq \llbracket\phi\rrbracket$, where we denote by $\llbracket S\rrbracket$, resp. $\llbracket\phi\rrbracket$, the prefix closed set of all finite traces of S, resp. $Spec_\phi$. Thus, since any trace $\langle a^i b ...\rangle$ with any number of actions a followed by an action b satisfies $\Diamond b$, the prefix closed set $\llbracket\Diamond b\rrbracket$ includes all the traces $\langle a^i\rangle$ with any number of actions a. Thus, we unavoidably have that a process S defined by $S = a \rightarrow S$ will satisfy $\llbracket S\rrbracket \sqsupseteq_{\mathcal{T}} \llbracket\Diamond b\rrbracket$ even though $S \not\models \Diamond b$ and $\llbracket S\rrbracket_\omega \not\subseteq \llbracket\phi\rrbracket_\omega$ (because $\langle a, a, a, \ldots\rangle \in \llbracket S\rrbracket_\omega$ and $\langle a, a, a, \ldots\rangle \notin \llbracket\Diamond b\rrbracket_\omega$). Similarly, for Q defined by $Q = a \rightarrow STOP$, we would have that $\llbracket Q\rrbracket \sqsupseteq_{\mathcal{T}} \llbracket\Diamond b\rrbracket$, even though $Q \not\models \Diamond b$. If we look at failure refinement with the same process $S = a \rightarrow S$ and formula $\llbracket\Diamond b\rrbracket$ where obviously $S \not\models \Diamond b$, we can see, as for trace refinement that $\llbracket S\rrbracket \sqsupseteq_{\mathcal{F}} \llbracket\Diamond b\rrbracket$

We have:

Proposition 1.

1. $S \models \phi \Rightarrow \llbracket S\rrbracket \sqsupseteq_{\mathcal{T}} \llbracket\phi\rrbracket$ *(and thus* $S \not\models \phi \Leftarrow \llbracket S\rrbracket \not\sqsupseteq_{\mathcal{T}} \llbracket\phi\rrbracket$*)* *but*
2. $S \models \phi \not\Leftarrow \llbracket S\rrbracket \sqsupseteq_{\mathcal{T}} \llbracket\phi\rrbracket$
3. $S \models \phi \not\Leftarrow \llbracket S\rrbracket \sqsupseteq_{\mathcal{F}} \llbracket\phi\rrbracket$

Proof: *The first proposition is straightforward. The previous (counter)-example proves part 2 and 3.*

It is thus *impossible* to achieve our goal in this manner, using the traces or failures refinements provided by CSP or FDR. We can actually pinpoint exactly when this approach fails and when it succeeds. We first present the following definition from [2]:

Definition 1. *Given a set S of traces in $\Sigma^\omega \cup \Sigma^*$ we define:* $pre(S) = \{\gamma \in \Sigma^* \mid \exists\sigma$ *with* $\gamma\sigma \in S\}$. *The LTL formula ϕ is a* liveness property *over an alphabet Σ iff* $pre(\llbracket\phi\rrbracket_\omega) = \Sigma^*$. *$\phi$ is a* safety property *over Σ iff* $\forall\gamma \in \Sigma^\omega$ *we have* $\gamma \not\models \phi \Rightarrow \exists\sigma \in pre(\{\gamma\})$ *such that* $\forall\delta$ $\sigma\delta \not\models \phi$.

[2] [18] defines a theory of infinite traces for CSP, but to our knowledge this has not been implemented in any tool for CSP. But even if FDR could handle such a theory of infinite traces, a proper encoding of $\llbracket\phi\rrbracket$ in CSP will in general be infinitely-branching (cf., Section 5), putting LTL model checking out of reach in practice.

Corollary 1. *Let ϕ be a liveness property. Then $[\![S]\!] \sqsupseteq_{\mathcal{T}} [\![\phi]\!]$ for any CSP process S and there exists a CSP process P such that $[\![P]\!] \sqsupseteq_{\mathcal{F}} [\![\phi]\!]$ and $P \not\models \phi$. Let ψ be a safety property and S a non-deadlocking CSP process. Then $S \models \psi$ iff $[\![S]\!] \sqsupseteq_{\mathcal{T}} [\![\psi]\!]$*

Any LTL property can be represented as the intersection of a liveness and a safety property [2] and our approach will fail for any LTL property which is not a pure safety property.

3.2 A more refined approach

Furthermore, even when using the full failure-divergence refinement combined with other CSP operators (such as hiding), we were unable to make this approach work in general! The main problem lies with the $\Diamond$ operator. Sometimes it is possible to find clever solutions (using hiding and divergence checking), but we found no solution which worked in general.

Example 3. For example, to check whether a divergence-free system S, satisfies $\Diamond b$ we could

- define $S' = S \backslash (\Sigma \setminus \{b\})$, i.e., hide all but the action b from S,
- then check whether $[\![b \rightarrow STOP]\!] \sqsupseteq_{\mathcal{T}} [\![S']\!]$, i.e., check that S' can perform b,
- and finally check that S' cannot diverge in the initial state, i.e., ensuring that b must eventually happen (this divergence test can be done using FDR).

It is thus possible, using hiding, traces refinement, and divergence testing to check whether a (divergence-free) system S satisfies $\Diamond b$.

Unfortunately, this approach (of using hiding plus divergence testing to test for eventuality $\Diamond$) does not scale up to more complicated formulas. E.g., when checking $\Box(a \Rightarrow \Diamond b)$, we can no longer systematically hide a; we would need to hide a (and check for divergence) after each occurrence of a so as to check whether $\Diamond b$ holds at that state. We thus searched for an alternate, generic solution of LTL model checking using the above ingredients augmented with more ingredients such as parallel composition and the failures semantics. Unfortunately, we were unable to come up with such an approach. (In fact, it might be impossible to do so.)

In summary, using existing features, it seems extremely difficult (maybe even impossible) for a FDR user to achieve LTL model checking of its CSP processes using the classical specification-based approach. In other words, the specification-based approach to verification, i.e., writing specifications and then checking whether your system is a valid refinement of that specification, does not seem to be very well suited for verifying temporal properties! (Maybe this situation will change if infinite traces [18] can be integrated into FDR.)

4 Model Checking Using a Tester and Composition

The unfruitful attempts in the previous section have led us to develop an alternate approach. Indeed, instead of checking whether a system S under consideration is a refinement of some specification ϕ, we can:

- build, from ϕ, a tester T_ϕ,
- then *compose* it with the system S, to finally
- check whether the composition satisfies some properties which ensure that $S \models \phi$.

If we look at the possible LTL formulas, for some of them a success or failure can be declared after having looked at a finite prefix of an infinite trace, such as $a, a \wedge b, \bigcirc a$. However, in general, entire infinite traces must be tested either to infer that a formula is satisfied (as in $\Box a$) or that it is not satisfied (as in $\Diamond a$). Therefore, a general solution is to build a tester whose behavior is similar for any formula.

A classical procedure defined by Vardi and Wolper [23] consists in verifying that $[\![S]\!]_\omega \cap [\![\neg\phi]\!]_\omega = \emptyset$ by building a system (Büchi automaton) able to do all the traces of $[\![\neg\phi]\!]_\omega$, composing it with S and verifying the emptiness of the resulting process using the Büchi acceptance condition. In brief, a Büchi automaton, is a finite automaton whose corresponding language is the set of infinite words which are accepted through a path which goes infinitely often on some accepting state.

We will try to pursue this avenue to solve our problem. In particular, we will use parallel composition to compose the system with the tester corresponding to $\neg\phi$.

However, we must take special care of deadlocking traces. Classically, when a system deadlocks, finite traces are extended by a special "Δ" (deadlock) action different from any others, so as to produce infinite traces only. Unfortunately, even though we can easily replace, in any CSP specification, $STOP$ by a process which loops on "Δ" actions, this is not possible in general. Take for example the process $(a \to b \to STOP) [\![\{a, b\}]\!] (a \to a \to STOP)$, where after the first a action, a deadlock occurs. No static analysis (not doing some kind of reachability analysis) is, for arbitrary CSP expressions, able to detect all such deadlocks. Moreover, since the system may be infinite state, in general this problem is clearly undecidable.

Therefore, since we do not want to (or cannot, e.g., wrt FDR) change the semantics of CSP (e.g., stipulating that when a process deadlocks it can perform Δ actions), we must consider a method which leaves the process S unchanged and build a tester which accepts both infinite traces (using Büchi acceptance condition) and deadlocking traces which satisfy the formula $\neg\phi$. The precise meaning of satisfaction of a formula by a deadlocking trace will be given later.

Therefore, in our setting 3 main problems arise:

1. how can we tackle deadlocking traces,
2. how can we translate the tester into CSP, and
3. how can we check emptiness using FDR.

We address all of these issues below.

4.1 Tackling Deadlocking Traces

To handle deadlocking traces we use LTL$^\Delta$ simply defined as LTL over $\Sigma \cup \{\Delta\}$ where $\Delta \notin \Sigma$ and where a valid trace π is either an infinite trace over Σ *or* a finite trace over Σ terminated with an infinite number of Δ's.

We have to be careful that the semantics of this extension is in agreement with our intuition. For example, intuitively a system S should satisfy $\neg a$ iff S can not perform an a as next visible action. Hence S may either perform only actions b different from a or it may deadlock! Similarly a system which satisfies $\neg \bigcirc a$ can either deadlock immediately or perform some visible action and then satisfy $\neg a$.

To capture our intuition about when a deadlocking trace satisfies an ordinary LTL formula over Σ, we can do a translation from LTL into LTL$^\Delta$, e.g., as follows:

- $\bigcirc\phi \quad \rightsquigarrow \quad \neg\Delta \wedge \bigcirc\phi$
- $\neg \bigcirc \phi \quad \rightsquigarrow \quad \Delta \vee \bigcirc\neg\phi$

The definition of $S \models \phi$ is very similar to the one for LTL:

$S \models \phi$ iff $\forall \pi \in [\![S]\!]_\Delta, \pi \models \phi$
where $[\![S]\!]_\Delta = [\![S]\!]_\omega \cup \{\gamma\Delta^\omega \mid (\gamma, \Sigma) \in \mathit{failures}(S)\}$, i.e., all the infinite traces of S plus all finite traces which can lead to a deadlock, then extended by Δ.

However, recall that even if satisfaction of LTL formulas by deadlocking traces is defined by extending these traces, in practice we have seen that we cannot modify the CSP system S to do the same. Therefore, we must build a tester which tests and accepts on the one hand, some infinite traces and on the other hand, deadlocking traces.

For that, we first use the classical construction of a Büchi automaton $\mathcal{B}$ for ψ, where ψ is the translation in LTL$^\Delta$ of $\neg\phi$ and ϕ is the LTL formula to check. This automaton $\mathcal{B}$ handles infinite traces from Σ^ω, but also (infinite) traces containing Δ actions.

Now, we know that the system S can only perform traces in $\Sigma^* \cup \Sigma^\omega$ and thus it is impossible to get traces which contain actions from Σ after an action Δ. We can use this to simplify $\mathcal{B}$. On the other hand, if $\mathcal{B}$ accepts a trace $\gamma\Delta^\omega$ where $\gamma \in \Sigma^*$, our tester should accept the finite trace γ if it is a deadlocking trace of S. To achieve this, we translate the Büchi automaton $\mathcal{B}$ into an extended automaton $\mathcal{B}_\Delta$ with *two* acceptance conditions:

- the classical Büchi acceptance condition for infinite traces
- another acceptance condition, based on a set of *deadlock monitor states*: a deadlocking trace γ will be accepted by $\mathcal{B}_\Delta$ if $\mathcal{B}_\Delta$ has a run taking the trace t which ends up in a so-called *deadlock monitor state*.

Definition 2. *A* Büchi Δ-automaton *is a six tuple $\mathcal{B} = (\Sigma, Q, T, Q^0, F, D)$ where Σ is the alphabet, Q is the set of states, $T \subseteq Q \times \Sigma \times Q$ is the transition relation, $Q^0 \subseteq Q$ is a set of initial states, $F \subseteq Q$ is a set of infinite trace accepting states, and $D \subseteq Q$ is a set of deadlock monitor states.*

Büchi Δ-automata include acceptance conditions both from finite automata and Büchi automata:

Definition 3. *Given a Büchi Δ-automaton $\mathcal{B} = (\Sigma, Q, T, Q^0, F, D)$, the language associated to $\mathcal{B}$ is $L(\mathcal{B}) = L_\omega(\mathcal{B}) \cup L_\Delta(\mathcal{B})$ with $L_\omega(\mathcal{B}) = \{\sigma | \sigma \in \Sigma^\omega$ and there are $s_0, s_1, s_2, ... \in Q$ and $\sigma = a_1, a_2, a_3, ...$ such that $s_0 \in Q^0$ and $s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} s_2$ ... and $s_i \in F$ for infinitely many values of $i\}$ and $L_\Delta(\mathcal{B}) = \{\sigma | \sigma \in \Sigma^*$ and there are $s_0, s_1, s_2, ...s_n \in Q$ with $s_n \in D$ and $\sigma = a_1, a_2, a_3, ...a_n$ such that $s_0 \in Q^0$ and $s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} s_2 ... \xrightarrow{a_n} s_n\}$*

In practice, we will modify a classical Büchi automaton $\mathcal{B}$ (over the alphabet $\Sigma \cup \{\Delta\}$) into a Büchi Δ-automaton $\mathcal{B}_\Delta$ (over the alphabet Σ)as follows

1. states, reachable from an initial state through transitions labelled by actions in Σ, which accept (with the classical Büchi condition) the string Δ^ω are defined to be *deadlock monitor states*,
2. all Δ transitions are now removed from $\mathcal{B}$,
3. transitions (and states) which cannot lead to the acceptance of a trace are removed

Observe that $L_\omega(\mathcal{B})$ is the language of $\mathcal{B}$ viewed as a classical Büchi automaton.

Note that in [22] Valmari defines a similar (though more sophisticated) tester.

One can easily see that the construction of $\mathcal{B}_\Delta$ from $\mathcal{B}$ can be done by an algorithm inspired from the Tarjan's search of strongly connected components (see, e.g., [20]): this algorithm does a linear parse of $\mathcal{B}$ (which defines the set of deadlock monitor states).

The algorithm can be found in Appendix A. This translation from $\mathcal{B}$ to $\mathcal{B}_\Delta$ is correct in the following sense:

Proposition 2. $L(\mathcal{B}_\Delta) = L_\omega(\mathcal{B}) \cap (\Sigma^\omega \cup \Sigma^*.\Delta^\omega)$.

The two following subsections discuss how to translate $\mathcal{B}_\Delta$ into CSP and how to check our two accepting conditions using FDR.

4.2 Translation of the Tester into CSP

We now present the translation of our Büchi Δ-automaton into a CSP process, which translates every state of Q into a CSP process and where

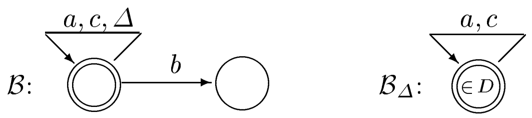
- an accepting state process produces a special *success* action (with $success \notin \Sigma$),
- for every deadlock monitor state a special "Δ" transition is added to the corresponding CSP process which leads to the special process $DEADLOCK$ defined below.

Definition 4. *Formally, we define our translation csp($\mathcal{B}$) of a Büchi Δ-automaton into a CSP process as follows:*

- *we map every $q \in Q$ to a CSP process name $NAME(q)$*
- *for every $q \in Q^0$ we add the CSP definition: $TESTER = NAME(q)$*
- *for every non-accepting state $q \in Q \setminus F$ and for all outgoing edges $(q, a, q') \in T$ we add the definition:*
  $NAME(q) = a \rightarrow NAME(q')$
- *for every accepting state $q \in F$ where $\{(q, a_1, q_1), \ldots, (q, a_n, q_n)\} \subseteq T$ are all the outgoing edges of q add the definition:*
  $NAME(q) = success \rightarrow (a_1 \rightarrow NAME(q_1) \,\Box\, \ldots \,\Box\, a_n \rightarrow NAME(q_n))$
- *For every state $q \in D$, we add a definition (this is equivalent to adding an external choice to the above definition):*
  $NAME(q) = deadlock \rightarrow DEADLOCK$
- *We add a single definition of $DEADLOCK$ (where $\Sigma = \{a_1, \ldots, a_n\}$):*
  $DEADLOCK = a_1 \rightarrow ko \rightarrow STOP \,\Box\, \ldots \,\Box\, a_n \rightarrow ko \rightarrow STOP$

The idea behind the special $DEADLOCK$ process is that if the system to be verified (with which it will run in parallel, synchronised on Σ) is not deadlocked then the $DEADLOCK$ process will be able to perform the special *ko* action (with $ko \notin \Sigma$). Hence, the existence of an accepted "really" deadlocking trace corresponds to a CSP failure trace $(deadlock, \{ko\})$ of $(S \,[\![\Sigma]\!]\, TESTER))\backslash(\Sigma \cup \{success\})$, i.e., we can perform *deadlock* and then *refuse* to perform the *ko* action.

Example 4. For $\phi = \neg\Diamond b$ and $\Sigma = \{a, b, c\}$ we would produce



and $csp(\mathcal{B}_\Delta) =$

$TESTER = State1$
$State1 = success \rightarrow ((a \rightarrow State1) \,\Box\, (c \rightarrow State1))$
$State1 = deadlock \rightarrow DEADLOCK$
$DEADLOCK = (a \rightarrow ko \rightarrow STOP) \,\Box\, (b \rightarrow ko \rightarrow STOP) \,\Box\, (c \rightarrow ko \rightarrow STOP)$

The above approach can easily be extended to CSP with datatypes, as provided by FDR. For example, if a is a channel of type *Int.Bool* and c a channel of type *Bool* we would produce:

$$State1 = success \rightarrow ((a?i?b \rightarrow State1) \Box (c?b \rightarrow State1))$$

One can also easily extend the basic propositions of LTL to enable more sophisticated pattern matching on actions. For example, one might want to check a formula $\Diamond a?i!true$ or $\Box(reqtoks?c?o \Rightarrow \Diamond colltoks!c!o?t)$ (see Appendix D).

4.3 Testing Emptiness in CSP/FDR

Let us summarise our approach so far. Given a CSP process S to be verified and an LTL formula ϕ to be checked, we do the following to construct a CSP process which will be used to verify $S \models \phi$:

1. negate the formula and translate it into LTL$^{\Delta}$, yielding ψ,
2. construct a Büchi automaton $\mathcal{B}$ for ψ using a classical construction,
3. translate $\mathcal{B}$ into a Büchi Δ-automaton $\mathcal{B}_\Delta$, to properly handle deadlocking traces,
4. translate $\mathcal{B}_\Delta$ into a CSP process $csp(\mathcal{B}_\Delta)$ (defining the $TESTER$ process).

We now want to check whether an infinite or a finite deadlocking trace of the system to be verified can satisfy $\neg\phi$. If no such a trace exists, then the formula ϕ is verified and the system is a model for ϕ. We conduct this test, using FDR, via two refinement checks: one for traces which generate an infinite number of successes and one to verify success due to deadlocks.

For the latter, we have already discussed in Subsection 4.2, that the existence of an accepted deadlocking trace corresponds to a CSP failure trace $(deadlock, \{ko\})$ of $D = (S \llbracket \Sigma \rrbracket TESTER)) \backslash (\Sigma \cup \{success\})$. To check this condition we thus simply use FDR to check whether $deadlock \rightarrow STOP \sqsupseteq_{\mathcal{F}} D$ holds.

The procedure to check the acceptance condition on infinite traces without deadlocks, looks like the one given in [23], except that our tester synchronised with the system will produce infinitely many *success actions* when it accepts a trace. More precisely, we have to check whether $S \llbracket \Sigma \rrbracket TESTER$ can produce a trace containing infinitely many *success* actions. This can be simplified into checking whether $C = (S \llbracket \Sigma \rrbracket TESTER) \backslash \Sigma$ can produce the infinite trace $success^\omega$. Now, as our environment (FDR) cannot analyse infinite traces, we resort to the following "trick": check using FDR whether $SUC \sqsupseteq_{\mathcal{T}} C$, where $SUC = success \rightarrow SUC$, i.e., checking whether for all i, $success^i$ can be done by C.

We would like to have $SUC \sqsupseteq_{\mathcal{T}} C$ iff $S \not\models \phi$. We will see that this depends on whether the system S is finite state or not.

Finite state processes Suppose that the system to be verified is finite state. Since the tester can also be defined as a finite state process, C will be finite state, and if C produces an unbounded number of success actions, it means that there must be a cyclic path, reachable from the initial state and including a *success* action. This is therefore equivalent to verifying that $success^\omega$ is a trace of C and we thus have the following proposition:

Proposition 3. *Let S be a finite state CSP process and ϕ a LTL formula. Let $TESTER$ by obtained by Def. 4. Then $S \models \phi$ iff $SUC \not\sqsupseteq_{\mathcal{T}} (S \llbracket \Sigma \rrbracket TESTER) \backslash \Sigma$.*

Note that one can put syntactic restrictions on the CSP processes to ensure that they are finite state (see, e.g., [16]): in our case it is sufficient to forbid any parallel operator in a recursive process.

Infinite state processes Let us show here on a counter example that in this case our verification procedure is *incomplete* (but still sound to conclude that the property indeed holds). Take the following CSP process definitions:

$S = (P \, [\![\{a, c\}]\!] \, Q) \; ||| \; R$, with
$P = a \to P \sqcap T$
$T = c \to T$
$Q = a \to (c \to STOP \; ||| \; Q)$
$R = b \to R$

The process R has been added to produce a non-deadlocking process. We can see that $S \models \neg\Box\Diamond c$, i.e. that S can never perform a c action forever, since in each branch of S, after a finite number of actions a and c, only b actions are possible. However, for each integer n, there is a branch (trace) which does n actions c. If we want to check if $S \models \neg\Box\Diamond c$ holds, the tester will produce a *success* action after each c action, and our test will conclude, since $SUC \sqsupseteq_{\mathcal{T}} C$, that $S \not\models \neg\Box\Diamond c$, which is wrong!

Notes:

- The previous example can also be used as a counter example to show that neither failure nor divergence will be able, in general, to discriminate an infinite trace from an unbounded one.
- The procedure given in [23] detects reachable loops. In general, this method will not be sufficient for infinite state systems. E.g. $S = a \to (S \, [\![\{a\}]\!] \, S)$ satisfies $\Box a$ but never loops! We suspect this problem to be undecidable.

4.4 Summary

To check $S \models \phi$ we perform the following 2 checks using FDR, where $TESTER$ by obtained by Def. 4:

1. $SUC \sqsupseteq_{\mathcal{T}} (S \, [\![\Sigma]\!] \, TESTER) \backslash (\Sigma \cup \{deadlock, ko\})$

2. $deadlock \to STOP \sqsupseteq_{\mathcal{F}} (S \, [\![\Sigma]\!] \, TESTER) \backslash (\Sigma \cup \{success\})$

If the first test succeeds, then we know, *if S is finite*, that $S \not\models \phi$ (there exists an infinite trace in S accepted by $csp(\mathcal{B}_\Delta)$). Otherwise, if the second test succeeds (there exists a deadlocking trace in S accepted by $csp(\mathcal{B}_\Delta)$), then $S \not\models \phi$. If both refinement checks fail, then we know that $S \models \phi$.

Observe that the first test uses the traces-refinement while the second one uses failures-refinement. This is because in the second test we have to check whether an "alleged" deadlock is a real deadlock.

A fully worked-out example in CSP (and FDR), checking whether a *System* satisfies $\neg\Diamond b$ is given in Appendix B. A more complicated and realistic example can be found in Appendix D, and the classical alternating-bit protocol is treated in Appendix C.

5 Preservation of LTL under Refinement

Despite the failure of refinement to capture temporal properties in Section 3, we can still derive some positive results. Suppose that for some LTL formula ϕ and CSP processe P, we know $Q \models \phi$ by applying the technique just presented. In addition, suppose that we derive a new CSP process Q which refines P, i.e., $P \sqsupseteq Q$: are there circumstances where we are assured that $P \models \phi$? A positive answer would allow us, in a design process where the consecutive specifications $S_0, S_1, \ldots, S_{n-1}, S_n$ satisfy $S_n \sqsupseteq S_{n-1} \ldots S_1 \sqsupseteq S_0$ and where we have checked $S_0 \models \phi$, to be sure that at the end $S_n \models \phi$. I.e., we would only have to model check the initial specification and not all the successive refinements. As we have already seen in Section 3, traces refinement alone is not sufficient to achieve this goal:

Proposition 4. *Traces refinement does not preserve satisfaction of LTL formulas.*

Proof. Using the following counter-example: $S_0 = a \rightarrow (b \rightarrow STOP \sqcap c \rightarrow STOP)$, $S_1 = a \rightarrow STOP \sqcap S_0$, $\phi \equiv \Diamond(b \vee c)$, we have $S_1 \sqsupseteq_{\mathcal{T}} S_0$, $S_0 \models \phi$ and $S_1 \not\models \phi$.

Unfortunately, the same holds for failures refinement in general:

Proposition 5. *Failures refinement does not preserve satisfaction of LTL formulas.*

Proof. Again, let us show that on a counter-example. In the paragraph 4.3 discussing infinite state processes, we have seen that for the formula $\phi \equiv \neg\Box\Diamond c$ and the infinite state process S, $S\backslash\{a,b\} \models \phi$ and $T = c \rightarrow T \sqsupseteq_{\mathcal{F}} S\backslash\{a,b\}$, but $T \not\models \phi$ (in fact $T \models \neg\phi$).

Fortunately, if we restrict ourselves to finite state processes or even finitely-branching processes, failures refinement does preserve LTL.

Proposition 6. *Failures refinement of finitely-branching CSP processes preserves satisfaction of LTL formula.*

Proof. Suppose that P and Q are finitely-branching CSP processes such that $P \sqsupseteq_{\mathcal{F}} Q$. We have to prove that for any LTL formula ϕ, $P \models \phi \Rightarrow Q \models \phi$, i.e. $[P]_{\Delta} \subseteq [Q]_{\Delta}$. Suppose $\gamma \in [P]_{\Delta}$ but $\gamma \notin [Q]_{\Delta}$. Either $\gamma \in \Sigma^{\omega}$ or $\gamma \in \Sigma^{*}\Delta^{\omega}$. **1.** If $\gamma \in \Sigma^{\omega}$ then Let τ be the tree representing the labelled transition systems of Q. We know that this tree is finitely-branching. Now, let us derive τ' from τ by removing from τ any node n (and its descendants) such that the path from the root of τ to n is not a prefix of γ. Trivially τ' is still finitely branching. We also know that τ' is infinite: suppose that τ' was finite then there must be maximum depth m of τ', which contradicts the fact that any prefix of γ (in particular the prefix of length $m+1$) can be generated by Q (because $P \sqsupseteq_{\mathcal{F}} Q$). Hence, by Königs lemma we know that there is an infinite branch in τ', i.e., $\gamma \in [Q]_{\Delta}$ which contradicts our hypothesis.

2. If $\gamma \in \Sigma^{*}\Delta^{\omega}$, then $\gamma = t\Delta^{\omega}$ and $(t, \Sigma) \in [P]$ and since $P \sqsupseteq_{\mathcal{F}} Q$, $(t, \Sigma) \in [Q]$. Therefore, $\gamma \in [Q]_{\Delta}$, which again contradicts our hypothesis.

Thus, if we manage write a finite state specification S_0 and model check a formula ϕ, then we do not have to check ϕ for refinements of S_0. Observe that this result, does not contradict Section 3 and does not enable us to solve the model checking problem itself more naturally using failures refinement! Indeed, a specification generating all traces and failures of an LTL formula ϕ will in general be infinitely branching (e.g., for $\Diamond b$) and we cannot apply Proposition 6. In fact, it will be finitely branching for safety properties but infinitely branching for liveness properties (cf. Corollary 1). So, even if the theory of infinite traces [18] were to be added to FDR, classical LTL model checking of finite systems, using refinement, will require the treatment of infinitely branching systems!

6 Complexity, Future Work, and Conclusion

LTL model checking is exponential in the size of the formula ϕ (PSPACE-complete [21]) and linear in the size of the system (NLOGSPACE).

At the complexity level, the difference between classical LTL model checking and our approach is due to the test of emptiness. For the method of Vardi and Wolper, it is linear in the size of the composition of the tester with the system and this composition can be done on-the fly [12]. Our procedure uses FDR to check language containment, whose complexity is PSPACE-complete (checking full failures/divergences refinement is even PSPACE-hard, even though "real" processes do not behave as badly [18]). On the other hand, using FDR means that optimisations such as hierarchical compression, data-independence and induction [19, 18, 9] can be applied. This will allow us to handle some infinite state systems, but the overall effect on the complexity is unclear (all the examples in the appendices were handled without any problem).

Fortunately, there is a way to get the best of both worlds, by adding a special check for refinement problems of the form $a^\omega \sqsupseteq_{\mathcal{T}} S$ into FDR, thus achieving the same linear complexity as Vardi and Wolper. Obviously, we cannot add this improvement ourselves to FDR, but we will try to convince the FDR implementors to do exactly that.

Another issue that needs to be resolved is the following: when a formula is *not* satisfied by a system, one of the interests of model checking is the production of a counter example. However, FDR only provides a counter example if a refinement check fails and not if the check succeeds; unfortunately the latter is what we would require! Fortunately, it seems possible to feed the result of the refinement checker into an animator (such as PROBE). Further work and cooperation with the FDR implementors is needed to establish this. Another interesting further research, is to study and apply our techniques within the context of other refinement-based formalisms such as action systems or B [1] and the associated tools B-TOOL and ATELIER-B. In fact, the refinement notion within these approaches connects more tightly with the infinite traces model of CSP than with the finite traces model [5], and so the relationship and needs will be somewhat different.

Other issues that should be studied further are the performance of (suitably extended) FDR on realistic benchmarks, and the study of other temporal logics such as CTL or CTL*. One possible approach to achieve CTL model checking of finite and infinite state CSP systems, is to write an interpreter for CSP in logic programming and then use the approach of [15]. Finally, we intend to find a semi-algorithm (using abstractions), to determine when an infinite state system does *not* satisfy a property.

History of the paper: This paper arouse out of discussions with proponents of refinement (and CSP) who proclaimed that "One does *not* need LTL model checking (for CSP or FDR), one can always write a specification describing all admissible behaviours and then checking that the system under consideration is a valid *refinement* of that specification." As we have shown in Section 3 this turns out to be extremely difficult (or even impossible in general). We have thus developed another, tester based approach, which can be fully automated. This paper thus underlines that "One *does* need LTL model checking, as one can *not* always (easily) write a specification describing all admissible behaviours and then checking that our system is a valid refinement of that specification."

In conclusion, we have have shed light on the relationship between model checking and refinement checking. We have unveiled shortcomings of the specification/refinement based approach to model checking, but have shown how to overcome them. Indeed, we have shown how to do LTL model checking of finite state CSP processes using refinement in general and the FDR environment in particular. We have also shown that our method is sound (but not complete) for processes which have an infinite number of states.

Acknowledgements

We want to thank Michael Butler, Marielle Doche, Javier Esparza, Andy Gravell, Ranko Lazic, Ulrich Ultes-Nitsche, Jean-François Raskin and Moshe Vardi for extremely useful comments, feedback, and explanations about CSP and LTL. We also would like to thank David Jackson for some insightful comments about FDR. Finally, we are grateful for the comments and constructive criticisms of the anonymous referees of VCL'2000 (a preliminary version of this paper appeared in the [informal] proceedings of VCL'2000).

References

1. J.-R. Abrial. *The B-Book*. Cambridge University Press, 1996.
2. B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, October 1985.
3. A.Pnueli. The temporal logic of concurrent programs. *Theoretical Computer Science*, 13:45–60, 1981.
4. R. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.
5. M. Butler and C. Morgan. Action systems, unbounded nondeterminism, and infinite traces. *Formal Aspects of Computing*, 7:37–53, 1995.
6. E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. The MIT Press, 1999.
7. E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.
8. E. M. Clarke and J. M. Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, Dec. 1996.
9. S. J. Creese and A. W. Roscoe. Data independent induction over structured networks. In *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA '00)*, Las Vegas, USA, June 2000.
10. J. Esparza. Decidability of model-checking for infinite-state concurrent systems. *Acta Informatica*, 34:85–107, 1997.
11. Formal Systems (Europe) Ltd. *Failures-Divergence Refinement — FDR2 User Manual*.
12. R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Proc. 15th Work. Protocol Specification, Testing, and Verification*, Warsaw, June 1995. North-Holland.
13. C. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
14. G. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1991.
15. M. Leuschel and T. Massart. Infinite state model checking by abstract interpretation and program specialisation. In A. Bossi, editor, Logic-Based Program Synthesis and Transformation. *Proceedings of LOPSTR'99*, LNCS 1817, pages 63–82, Venice, Italy, September 1999.
16. J. Magee and J. Kramer. *Concurrency: State Models & Java Programs*. Wiley, 1999.
17. K. L. McMillan. *Symbolic Model Checking*. PhD thesis, Boston, 1993.
18. A. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1997.
19. A. W. Roscoe and R. S. Lazic. Using logical relations for automated verification of data-independent CSP. In *Proceedings of Oxford Workshop on Automated Formal Methods ENTCS*, 1996.
20. R. Sedgewick. *Algorithms in C++*. Addison-Wesley, 1992.
21. A. P. Sistla and E. M. Clarke. The complexity of propositional linear temporal logics. *Journal of the ACM*, 32(3):733–749, July 1985.

22. A. Valmari. On-the-fly verification with stubborn sets. In C. Courcoubetis, editor, *Proceedings of the International Conference on Computer-Aided Verification (CAV'93)*, LNCS 697, pages 397–408. Springer-Verlag, 1993.
23. M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proceedings of LICS'86*, pages 332–344, 1986.

A Algorithm to compute deadlock monitor states

The following algorithm takes a Büchi automaton from a LTL^{Δ} formula and defines the set of *deadlock monitor states*.

By definition a state s is a deadlock monitor state if it is possible, starting from s, to take a path of Δ transitions which goes infinitely often to a (Büchi) accepting state. Since in our case, the Büchi automaton is finite, the previous condition is equivalent to the following: using Δ transitions, it is possible to reach from s a cycle of Δ transitions which contains a (Büchi) accepting state.

The following algorithm is an extension of the search for the strongly connected components (*scc*) from Tarjan (see e.g. [20]). It does a depth first search using only Δ transitions. It detects cycles (of Δ transitions) containing a Büchi accepting state to identify new deadlock monitor states. Then it synthesizes the information (whether we can reach a cycle with some accepting state) up to the "origin" of a strongly connected component, and then updates the information for all states in this *scc*.

Our recursive algorithm supposes that the automaton has n states and that each state has an entry in the following arrays:
- *accept*: boolean: says if the state is (Büchi) accepting.
- *nb*: integer, gives its number in the parsing,
- *dms*: boolean, true if the state is a 'deadlock monitor state'.

The return value is the minimal number one can reach trough the state. *nb* and the return value are handled as in the standard algorithm of Tarjan. *dms* and its handling have been added. Moreover, the (local) integer variable *Acc* keeps the number of the last accepting state which is currently on the stack: it is used to detect the cycles of (Δ) transitions which contain an accepting state.

x, y *Acc*, *min* and i are local to the recursive function *scc*; the graph (Büchi automaton) with the arrays *nb*, and *dms* are global to the algorithm. The stack S and the variable *cpt* are also global.

```
//initialisations
cpt := 0;
P <= Empty;
for i := 1 to n do nb[i] := 0;
for i := 1 to n do
   if nb[i] = 0 then  // state i not parsed yet
      m := scc(i,0);  // m is not used here
   fi
od

integer function scc(x, Acc : integer)
var
   m,y,i,min: integer;
begin
   cpt := cpt+1;
   nb[x] := cpt;                    //parsing number of x
   min := cpt;
   Push(S,x);
   if accept[x] then
      Acc = cpt;       // Acc keeps the biggest nb among the states in
                              parsing which are accepting
```

```
  fi
  for all y such that <x, Delta, y> is a transition do  // parses its neighbours
     if nb[y] = 0 then               // not yet parsed
        m := scc(y,Acc);
        if m  < min then
           min := m;                 // new minimal state reachable through y
        fi
        dms[x] := dms[x] or dms[y];  // dms[x] is now maybe true thanks to y
     else                            // y parsed or in parsing
        if dms[y] then
           dms[x] := true;           // dms[x] is now true thanks to y
        else
           if Acc > nb[y]      // dectection of a cycle with an accepting state
              dms[x] := true;
           fi
        fi
        if nb[y] < min then      // y is in parsing and is a new minimum for x
           min := nb[y];
        fi
     fi
  od
  if min = nb[x] then               // x is an ``entry point'' of a s. c. c.
     while Top(S) <> x do           // update all states is the same s.c.c.
        y := Top(S);
        nb[y] := n+1;               // = mark y parsed
        dms[y] := dms[x];
        Pop(S);
     od
     nb[x] := n+1;
     Pop(S);
  fi
  scc := min;                       // return the value min
endfunction
```

At the end of this algorithm, the modified Büchi automaton can easily be created by removing Δ transitions (and maybe cleaning it by keeping only useful states).

B A Simple Example in FDR 2.28

Here is an original System to verify, using the machine-readable CSP syntax employed by FDR:

```
channel a,b,c,d
System =  (b->System) []  (b->c->a->SKIP)
```

Suppose we wanted to establish whether the system satisfied $\Diamond b$ using $\Sigma = \{a, b, c, d\}$. We would then construct the extended Büchi automaton $\mathcal{B}$ for $\neg\Diamond b$ (see Ex. 4) and apply our translation (Def. 4) to obtain $csp(\mathcal{B})$:

```
channel success, deadlock,ko
TESTER = State1
State1 = (success->((a->State1) [] (c->State1) [] (d->State1))
State1 =  deadlock -> Deadlock
Deadlock = (a->ko->STOP) [] (b->ko->STOP) [] (c->ko->STOP) [] (d->ko->STOP)
```

We now compose $csp(\mathcal{B})$ with the system to be verified:

```
Composition = (System [| {a,b,c,d} |] TESTER) \{a,b,c,d,deadlock,ko}
```

and then check whether this composition can generate an infinite number of success actions:

```
SUC = success -> SUC

assert Composition [T= SUC
```

In our case, this refinement test fails. However, for *System2* defined below, it succeeds, meaning that *System2* does not satisfy $\Diamond b$ (as it is finite state):

```
System2 =  (a->System) []  (b->c->a->SKIP) [] STOP
```

To test whether there is, in our *System*, a deadlocking trace that accepts the formula $\neg\phi$ we do the following composition:

```
Composition2 = (System [| {a,b,c,d} |] TESTER) \{a,b,c,d,success}
```

and check whether this can generate a real deadlock:

```
RealDeadlock = deadlock -> STOP

assert Composition2 [F= RealDeadlock
```

In our case, this refinement test fails and we have thus established $System \models \Diamond b$. However, for $System3$ and $System4$ defined below, it succeeds, meaning that the systems do not satisfy $\Diamond b$:

```
System3 =  (a->c->a->STOP)
System4 =  (b->System) []  (b->c->a->SKIP) [] STOP
```

C The alternating-bit protocol in FDR

In the following we have modelled and verified the classical alternating bit protocol, to see how a well known example is represented in our approach (and how FDR performs on it).

```
ChData = {0,1,2,3}
ChControl = {0,1,2}
channel in,outb: ChData
channel out,inb: ChData.ChControl
channel ok,err

StartA(x) = AS1((x+1)%4) -- description of process A, the sender
AS1(x) = out!x!1 -> AS2(x)
AS2(x) = in?vr ->
             if (vr==1) then  StartA(x)
                else (if (vr==0) then AS3(x)
                   else AS5(x))
AS3(x) = out!x!1 -> AS2(x)
AS4(x) = in?vr ->
             if (vr==1) then  StartA(x)
              else if (vr==0) then StartA(x)
                 else AS5(x)  -- error
AS5(x) = out!x!0 -> AS4(x)

-- description of process B, the receiver, BS2 is the initial state
Breceived(x,y) = if (x==y) then  ok -> BS1((y+1)%4)
                           else  err -> BS1((y+1)%4)
BS1(y) = outb!1 -> BS2(y)
BS2(y) = inb?x?ar ->
             if ar==1 then  Breceived(x,y)
              else if ar==0 then BS3(y)
                 else BS5(y)  -- error

BS3(y) = outb!1 -> BS2(y)
BS4(y) = inb?x?ar ->
             if ar==1 then  Breceived(x,y)
     else if ar==0 then Breceived(x,y)
          else BS5(x)  -- error
BS5(y) = outb!0 -> BS4(y)
```

```
Buff2(cin,cout) = cin?x -> Buff2m(cin,cout,x)
Buff2m(cin,cout,x) = (cout!x -> Buff2(cin,cout))
                   [] (cin?y -> cout!x -> Buff2m(cin,cout,y))
Buff22(cin,cout) = cin?x?y -> Buff22m(cin,cout,x,y)
Buff22m(cin,cout,x,y) = (cout!x!y -> Buff22(cin,cout))
                   [] (cin?u?v -> cout!x!y -> Buff22m(cin,cout,u,v))

System = (((StartA(0) [| {| out |} |] Buff22(out,inb)) \{|out|})
             [|  {| inb, in |} |]
          ((BS2(1) [| {| outb |} |] Buff2(outb,in)) \{|outb|}))
```

Let us now check the liveness property $\Box\Diamond ok$:

```
-- Verify property: always eventually ok
-- 1. Buechi automata translation for  not Alw Ev ok = Ev Alw not ok
channel success,deadlock,ko
TESTER = State1
State1 =  (in?x->State1 [] ok->State1 [] err->State1 [] inb?x?y->State1)
           |~| (State2)
State2 =  (success->(in?x->State2 [] err->State2 [] inb?x?y->State2))
          [] deadlock -> Deadlock

Deadlock =  (in?x->ko->STOP [] ok->ko->STOP [] err->ko->STOP [] inb?x?y->ko->STOP)

-- 2. compose system with TESTER
Composition = (System [| {|in,ok,err,inb|} |] TESTER)
                    \{|in,ok,err,inb,deadlock,ko|}
DComposition = (System [| {|in,ok,err,inb|} |] TESTER)
                    \{|in,ok,err,inb,success|}

-- 3. check whether intersection empty
SUC = success->SUC
assert Composition [T= SUC
RealDeadlock = deadlock->STOP
assert DComposition [F= RealDeadlock
```

Both of these tests fail (instantaneously) using FDR, i.e., the system satisfies the liveness property $\Box\Diamond ok$. For example to check `assert Composition [T= SUC` we get:

```
...
-- Normalisation complete
Starting timer
About to start refinement
Refinement check: *
+.*
+.*
+.*
+.Trace error after 4 states

Refine checked 4 states
With 3 transitions

Found 1 example
Stopped timer
Resource   Start              End                Elapsed
Wall time    184906             184906                 0
CPU (self)        7                  7                 0
CPU (sys)         1                  1                 0
(inc children)
CPU (self)       16                 16                 0
CPU (sys)         2                  2                 0

Started: 0 Finished: 0 Transitions: 0

Ready
Started: 28 Finished: 28 Transitions: 68

Ready
```

The following safety property ($\Box\neg err$) can also be verified instantaneously:

```
-- A safety property:  always not err
TESTER2 =  (in?x->TESTER2 [] ok->TESTER2 [] err->State22 [] inb?x?y->TESTER2)
State22 =  success->State22
Composition2 = (System [| {|in,ok,err,inb|} |] TESTER2) \{|in,ok,err,inb|}
assert Composition2 [T= SUC
-- test fails -> safety property ok
```

If we now use a faulty buffer, which can (repeatedly) loose messages the safety property still holds but the liveness property is now violated:

```
-- Now the system with a faulty buffer

FaultyBuff2(cin,cout) = cin?x -> FaultyBuff2m(cin,cout,x)
FaultyBuff2m(cin,cout,x) = (cout!x -> FaultyBuff2(cin,cout))
                   [] (cout!2 -> FaultyBuff2(cin,cout))
                   [] (cin?y -> (   (cout!x -> FaultyBuff2m(cin,cout,y))
                                 [] (cout!y -> cout!x -> FaultyBuff2(cin,cout))))

FaultySystem = (((StartA(0) [| {| out |} |] Buff22(out,inb)) \{|out|})
             [|  {| inb, in |} |]
          ((BS2(1) [| {| outb |} |] FaultyBuff2(outb,in)) \{|outb|}))
FaultyComposition = (FaultySystem [| {|in,ok,err,inb|} |] TESTER)
                   \{|in,ok,err,inb,deadlock,ko|}
assert FaultyComposition [T= SUC
-- test succeeds: liveness property violated !
DFaultyComposition = (FaultySystem [| {|in,ok,err,inb|} |] TESTER)
                   \{|in,ok,err,inb,success|}
assert DFaultyComposition [F= RealDeadlock
-- test fails

FaultyComposition2 = (FaultySystem [| {|in,ok,err,inb|} |] TESTER2) \{|in,ok,err,inb|}
assert FaultyComposition2 [T= SUC
-- test fails: safety property ok
```

D A more complicated example in FDR

The following is a more complicated CSP specification, which models distributed system for pension (i.e., tokens) distribution via postoffices. Customers have a preferred postoffice, but they can collect their pension from any postoffice. In the latter case, the authority to distribute the pension must be requested from the Home-postoffice. (This case study grew out of interaction with one of the industrial partners of the EPSRC-funded critical systems project "ABCD", currently ongoing at the University of Southampton.)

```
-- =========================================================
-- A distributed pension distribution scheme via Postoffices
-- by Marielle Doche, University of Southampton
-- =========================================================
nametype Tokens = {0..5}
nametype Cust = {1,2,3}  -- 3 customers
nametype Home = {0,1,2}  -- 0 = centre; 1,2 = offices
nametype Office = diff(Home, {0})
home(1) = 1
home(2) = 2
home(3) = 0
channel reqtoks : Cust.Office
channel colltoks : Cust.Office.Tokens

-- This process gives a global specification from a Customer point of view
CUST(c,n) = reqtoks.c?o -> colltoks.c!o!n -> CUST(c,0)

-- abstract spec of the system with 3 customers
SPEC = ||| c : Cust @ CUST(c,4)
-----------------------------------------------------------------------------
-- This specification describes the centre, which communicates with the offices
-- the new channels are
channel disthome, sendoff, rechome : Cust.Office.Tokens
```

```
channel reqoff, queryhome : Cust.Office
CENTRE(c,n) =
     n>0  and home(c)!=0 & disthome.c?o:{home(c)}!n -> CENTRE(c,0)
   []
     reqoff.c?o:Office ->
          (if n>0 or home(c)==0 or home(c)==o
               then sendoff.c!o!n -> CENTRE(c,0)
               else queryhome.c?o1:{home(c)} -> rechome.c.home(c)?a:Tokens
                            -> sendoff.c!o!a -> CENTRE(c,n) )
-----------------------------------------------------------------------------
-- This specification describes an office which communicates with the centre
-- about a customer
-- the new channels to or from the centre
channel sendcentre, reccentre, recdist : Cust.Office.Tokens
channel reqcentre, querycentre : Cust.Office
OFF(c,o,n) =
     n==0 & recdist.c.o?a:Tokens -> OFF(c,o,a)
   []
     reqcentre.c.o -> sendcentre.c.o!n -> OFF(c,o,0)
   []
     reqtoks.c.o ->
          (n > 0 & colltoks.c.o!n -> OFF(c,o,0)
        []
          n ==0 &  (
               (querycentre.c.o -> (
                    reccentre.c.o?a:Tokens -> colltoks.c.o!a -> OFF(c,o,0)
                  []    -- (+)
                    recdist.c.o?a:Tokens ->
                    reccentre.c.o?b:Tokens ->      -- ($)
                    -- colltoks.c.o!a -> OFF(c,o,0)   -- (+)
                    colltoks.c.o!a -> OFF(c,o,b)   -- (+) ($)     ) )

             []
               (o == home(c) & recdist.c.o?a:Tokens
                    -> colltoks.c.o!a -> OFF(c,o,0))
          ) )
-----------------------------------------------------------------------------
-- This process describe for a given customer a synchronous communication
-- between the centre and the offices
SYNCHCOM(c,n) =
     CENTRE(c,n)
     [disthome.c.o.a <-> recdist.c.o.a,
  sendoff.c.o.a <-> reccentre.c.o.a,
  rechome.c.o.a <-> sendcentre.c.o.a,
  reqoff.c.o <-> querycentre.c.o,
  queryhome.c.o <-> reqcentre.c.o |
  o <- Office, a <- Tokens]
     (|||o: Office @ OFF(c,o,0))
-----------------------------------------------------------------------------
SYNCHTRANS(c,n) =
     CUST(c,n)[|{|reqtoks.c, colltoks.c |}|]
     (SYNCHCOM(c,n)\{|disthome.c, recdist.c, sendoff.c,
                 reccentre.c, rechome.c, sendcentre.c,
                 reqoff.c, querycentre.c, queryhome.c, reqcentre.c|})
SYNCH = ||| c : Cust @ SYNCHTRANS(c,4)
```

In the remainder of this appendix, we will assume that any channel or action stands for all its “completions”. E.g., *reqtoks* stands for *reqtoks*.1.0, *reqtoks*.1.1, ….

Let us now try to verify the LTL formula $\Box(reqtoks \Rightarrow \Diamond colltoks)$, i.e., whenever a token (i.e., pension) is requested by a user a token will eventually be collected. For this we first negate the formula, i.e., we get $\neg\Box(reqtoks \Rightarrow \Diamond colltoks) = \Diamond(reqtoks \wedge \neg\Diamond colltoks)$ $= \Diamond(reqtoks \wedge \Box\neg colltoks)$. We now translate this into a Büchi automaton, and simplify for deadlocks, giving us Figure 1. (Observe that this automaton is non-deterministic; there is no equivalent deterministic automaton for that property.) We now translate Figure 1 into CSP as described in the paper:

```
channel success,deadlock,ko
TESTER = STATE1
```

```
STATE1 =  reqtoks?c?o -> STATE1 [] colltoks?c2?o2?t -> STATE1
          [] reqtoks?c?o -> STATE2
STATE2 = ((success -> (reqtoks?c?o -> STATE2))
          [] deadlock -> Deadlock)
Deadlock = (reqtoks?c?o->ko->STOP [] colltoks?c?o?t->ko->STOP)
```

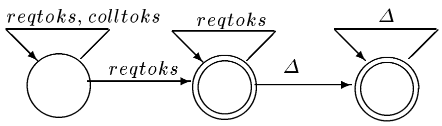


Fig. 1. A Büchi automaton for $\Diamond(reqtoks \wedge \Box\neg colltoks)$

We now encode our refinement checks as described in the paper:

```
SComposition = (SPEC [| {|reqtoks,colltoks|} |] TESTER) \{| reqtoks,colltoks,deadlock,ko|}
SComposition2 = (SPEC [| {|reqtoks,colltoks|} |] TESTER)\{| reqtoks,colltoks,success |}
Composition = (SYNCH [| {|reqtoks,colltoks|} |] TESTER) \{| reqtoks,colltoks,deadlock,ko |}
Composition2 = (SYNCH [| {|reqtoks,colltoks|} |] TESTER)\{| reqtoks,colltoks,success |}
SUC = success->SUC
RealDeadlock = deadlock->STOP
assert SComposition [T= SUC
-- refinement fails => no infinite trace violates formula => OK
assert SComposition2 [F= RealDeadlock
-- refinement fails => no deadlocking trace violates formula => OK
assert Composition [T= SUC
-- refinement fails => no infinite trace violates formula => OK
assert Composition2 [F= RealDeadlock
-- refinement fails => no deadlocking trace violates formula => OK
```

So, both the very high-level specification $SPEC$ and the more detailed specification $SYNCH$ satisfy the LTL formula $\Box(reqtoks \Rightarrow \Diamond colltoks)$.

We can actually try to verify a more complicated property, namely $\Box(reqtoks?c?o \Rightarrow \Diamond colltoks!c!o?t)$, by re-defining STATE1 as follows:

```
STATE1 =  reqtoks?c?o -> STATE1 [] colltoks?c2?o2?t -> STATE1
          [] reqtoks?c?o -> STATE2(c,o)
STATE2(c,o) = ((success -> ((reqtoks?c2?o2 -> STATE2(c,o))
                   [] (colltoks?c2?o2?t -> STATE3(c,o,c2,o2)))) [] deadlock -> Deadlock)
STATE3(c,o,c2,o2) =
          if ((c==c2) and (o==o2)) then STOP else STATE2(c,o)
```

We are now checking that if a customer c request a token (at office o) that he will eventually get a token (at that same office o).

Now the refinement checks look as follows:

```
assert SComposition [T= SUC
-- refinement succeeds => infinite trace violates formula => NOT OK !!
assert SComposition2 [F= RealDeadlock
-- refinement fails => no deadlocking trace violates formula => OK

assert Composition [T= SUC
-- refinement succeeds => infinite trace violates formula => NOT OK !!
assert Composition2 [F= RealDeadlock
-- refinement fails =>  no deadlocking trace violates formula => OK
```

This is essentially due to possible starvation of a customer because other customers can repeatedly ask for and always collect a token before he gets his token. However, if we change the specification of the behaviour of customers to:

```
CUST(c,n) = reqtoks.c?o -> colltoks.c!o!n -> STOP
```

i.e., the customers do not repeatedly ask for tokens (pensions), then all refinement checks fail and the formula is actually satisfied.