Watchdog Transformations for Property-Oriented Model-Checking

Michael Goldsmith[1,4], Nick Moffat[2], Bill Roscoe[3], Tim Whitworth[1], and Irfan Zakiuddin[2]

[1] Formal Systems (Europe) Ltd.
{michael, tim}@fsel.com
[2] Systems Assurance Group, QinetiQ, Malvern, UK
{N.Moffat, I.Zakiuddin}@eris.QinetiQ.com
[3] Computing Laboratory, University of Oxford, UK
Bill.Roscoe@comlab.ox.ac.uk
[4] Worcester College, Oxford, UK

Abstract. We discuss how to transform a CSP refinement, $S \sqsubseteq I$, to enable all its events to be hidden; this is useful because many of the state space compression functions provided by the model-checker FDR are effective only when events are hidden [1]. In an earlier paper [2] we described a suitable transformation for the case where the refinement is in the traces semantics of CSP. This paper extends the approach to the more difficult case of the stable-failures semantics. In both cases, a watchdog transformation is applied to the specification S, resulting in a *watchdog process* WD_S, which is then composed in parallel with I, or with I in a simple context. The watchdog process monitors I and somehow indicates whether it can behave in a way that is incompatible with refinement of S. All events of the original assertion can be hidden in the transformed assertion. We also discuss the design of compression strategies that try to hide as many events as possible in the component processes of I and WD_S, and compress the composition as it is being built up. We describe our implementation of the watchdog transformations and some simple compression strategies.
Keywords: Compression, CSP, FDR, Model-Checking, State Explosion Problem, Watchdog Transformation.

1 Introduction

It is widely recognised that the state explosion problem limits the tractability of model-checking. There are several approaches to combating state explosion, perhaps the best known of which are compositional reasoning [3], abstraction [4], symmetry exploitation [5–7], and partial order reduction [8–10]. Symbolic model-checkers [11] use data structures, often BDDs, for efficiently representing the explored state space and the state transition function. An alternative approach is taken by the FDR [12] model-checker for CSP. FDR stores the set of explored states explicitly, but provides compression functions that approximate semantic minimisation, reducing the size of the (internal) state machine for

a process, without changing its behaviour. Use of these compression functions can dramatically improve the tractability of CSP model-checking [13]. Property-oriented model-checking is a novel technique which uses the property – the specification process and the semantic model of refinement – to maximise the benefit of compression.

We discuss how to transform a CSP refinement, $S \sqsubseteq I$, to enable all its events to be hidden. This is useful because FDR's compression functions are most effective when events are hidden [1]. In an earlier paper [2] we described a suitable transform for the simple case where the refinement is in the traces semantics of CSP. This paper extends the approach to the more difficult case of the stable-failures semantics. In both cases, a watchdog transformation is applied to the specification process S, resulting in a watchdog process WD_S which is then composed in parallel with I, or with I in a simple context. In a suitable watchdog assertion, the watchdog process monitors I and somehow indicates whether it can behave in a way that is incompatible with refinement of S. We ensure that the truth or falsity of the assertion is preserved when all events of the original assertion are hidden in the watchdog assertion.

The traces watchdog process and watchdog assertion are summarised in Section 2. A common special case of the watchdog transformation for the more difficult case of the stable-failures semantics is presented in Section 3. This restriction is removed in Section 4, which presents the general transformation. These sections informally prove correctness (in a sense made precise later).

It is one thing to hide a given set of process events and then compress, and another to hide and compress efficiently. FDR operates on processes expressed in $\mathrm{CSP_M}$ [14], the machine-readable version of CSP. FDR's compression functions generally explore the full state space of a $\mathrm{CSP_M}$ process in order to derive a compressed state machine, so naively hiding all possible events and then compressing would cause FDR to traverse the full state space. Section 5 describes compression strategies that aim to achieve closer to optimal compression given a set of events to be hidden. These compression strategies try to reorganise the $\mathrm{CSP_M}$ process structure to allow events to be hidden as early as possible when combining component processes of I and WD_S, and thus compress the composition as it is being built up.

Section 6 discusses the implementation of the watchdog transformations and of some simple compression strategies. We end with a discussion in Section 7 of the relationship to other work.

Throughout, we use the notational convention that $\alpha(P)$ denotes the complete alphabet of a process P, and $\alpha(P, Q)$ denotes $\alpha(P) \cup \alpha(Q)$.

1.1 The Watchdog Approach

The basic technique consists of three distinct steps, the first of which is performed in a way that guarantees that the second is possible:

1. Transform $S \sqsubseteq I$ into an equivalent assertion $P(\alpha(S, I)) \sqsubseteq F(WD_S, I)$ in which P is a function from alphabets to processes, and watchdog process WD_S is composed with I using some composition function F.

2. Without affecting its truth, transform this assertion to one where $\alpha(S, I)$ is hidden on both sides: $P(\alpha(S, I)) \setminus \alpha(S, I) \sqsubseteq F(WD_S, I) \setminus \alpha(S, I)$.

3. Simplify the left-hand-side and apply a compression strategy to generate a compressed state machine for the right-hand-side $F(WD_S, I) \setminus \alpha(S, I)$.

2 Watchdog Transformations for the Traces Semantics

The watchdog (process) transformation described in [2] maps a specification process, S, to a watchdog process that monitors the traces of an implementation process, I, and indicates whether or not S is refined by I according to CSP's traces semantics.

The watchdog process, WD_S in Section 1.1, is defined so that it can perform a distinguished *fail_* event when I performs a trace not permitted by S. This transformation is most easily defined in terms of a normal form definition of S. This is unsurprising: the efficient checking of refinement within FDR relies upon normalisation of the specification – determinising the underlying state machine, while producing sufficient annotations to allow any interesting nondeterminism to be reconstructed. Allowed behaviour of the implementation can then be verified by a local check. Also, a normal form generally has a simple structure (relying on a restricted set of process operators) which allows a simple definition of the watchdog process. We use the traces normal form defined in [1] since this is the one implemented within FDR. This allows the transformation to be implemented both directly and efficiently, as explained below.

In the remainder of this section, we describe the traces normal form, then the watchdog transformation for traces refinement, then the watchdog assertion, and finally we argue that the watchdog assertion holds iff the original assertion holds.

2.1 The Traces Normal Form

Any process P has a trace-equivalent expression as an entry into a mutual recursion of the following form (for some indexing set and functions A and *after*):

$$P'(i) = \Box\ a \in A(i) \bullet a \rightarrow P'(after(i, a)).$$

For example, we may take the indexing set to be the set of all traces of P; $A(s)$ to be $inits(P/s)$, the set of all events that P can perform after the trace s[1]; and $after(s, a)$ to be the extended trace $s^\frown\langle a\rangle$. In general, though, that particular construction may give rise to an infinite mutual recursion even for very simple processes.

P is finite-state (in the traces model) precisely if there is such a representation with a finite indexing set, and this representation is a traces normal form if there

[1] The set returned by the function *inits* is often referred to as the *initials* of its argument. Thus, $inits(P/s)$ is the initials of P after it has performed trace s.

is no non-trivial bisimilarity between terms in the recursion. Each set $A(i)$ is the union of the initial events of all operational states reachable on any trace that leads to $P'(i)$.

2.2 The Traces Watchdog Process

Given a specification $S = S'(i_0)$ as such a (finite) traces normal-form recursion, we can define a watchdog process, $WDT_S(i_0)$, in terms of a transform WDT_S defined by recursion over the same index set:

$$\begin{aligned} WDT_S(i) = (&\Box\ a \in A(i) \bullet a \rightarrow WDT_S(after(i,a))) \\ &\Box \\ &(\Box\ b \in \alpha(I) - A(i) \bullet b \rightarrow fail\_ \rightarrow STOP) \end{aligned}$$

Note that $A(i)$ is $inits(S/s_i)$ for any trace s_i that takes S to the state indexed by i, and *inits* is a semantic function that is not expressible in the $\mathrm{CSP_M}$ language; so $A(i)$ must be calculated (using FDR) as part of the transformation.

The intention is that $WDT_S(i_0)$ can perform any trace tr that S can perform, but it can also perform events from the alphabet of I not allowed by S/tr (after which it can only perform the $fail\_$ event).

Notice that this definition of WDT is expressed in terms of the alphabet, $\alpha(I)$, of the implementation process I. Similar formulations independent of $\alpha(I)$ are possible, but they require a slightly more complex composition function F. We use the formulation above for simplicity.

2.3 The Traces Watchdog Assertion

Having calculated the traces watchdog process $WDT_S(i_0)$, the transformed assertion is

$$RUN(\alpha(S,I)) \quad \sqsubseteq_{\mathcal{T}} \quad I \underset{\alpha(S,I)}{\parallel} WDT_S(i_0) \tag{1}$$

in which I synchronises with the watchdog process on $\alpha(S,I)$, the events of the original assertion. The process $RUN(\alpha(S,I))$ has a single state, in which it can perform any event of the original assertion (and return to the same state).

2.4 Correctness

We say that a watchdog assertion transformation for the traces semantics is correct if it preserves the truth of a traces assertion. In [2] it is proved that the transformation described in Section 2.3 (defined in terms of WDT) is correct in a stronger sense which, in addition, relates counterexamples of the watchdog assertion to corresponding counterexamples of the original assertion. We do not describe this stronger property here, due to lack of space.

Essentially, if I can only perform traces of S, then $WDT_S(i_0)$ is constrained by I to traces of S, so can never progress via its second branch. So the parallel composition of I and $WDT_S(i_0)$ can never perform the $fail\_$ event. Conversely,

if this composition can perform $fail\_$, then it must be performed by $WDT_S(i_0)$ via its second branch, in particular after some event e outside $inits(S/s)$, where s is some trace of S – that is, there must be a trace $s^\frown\langle e\rangle$ of I that is not a trace of S.

Notice that $RUN \sqsubseteq_{\mathcal{T}}$ is invariant under hiding on both sides when the set hidden is contained in the argument of RUN. Therefore the traces assertion transformation described allows events of $\alpha(S, I)$ to be hidden on both sides without affecting truth or falsity. Moreover, FDR's debugger can 'look inside' process operators, in particular the hiding operator; this allows the user to find a counterexample of the original assertion corresponding to any given counterexample of the watchdog assertion.

3 Watchdog Transformations for the Failures Semantics

In this section we introduce a stable-failures model watchdog transformation that is valid for a common special case of S and I. Section 4 describes the unrestricted form.

It is worth reviewing CSP's stable-failures model briefly. The *failures* of a process P, $failures(P)$, is the set of all possible observations (s, X) of a trace s of P leading to a stable state that can refuse to engage in any event in X, a so-called refusal (set) of P/s. (A *stable state* is one that cannot perform internal actions.) Then, a refinement $P \sqsubseteq_{\mathcal{F}} Q$ holds in the stable-failures model if $traces(Q) \subseteq traces(P)$ and $failures(Q) \subseteq failures(P)$. So failures refinement implies traces refinement[2]. Additionally, failures refinement requires that the implementation can never (stably) refuse more events after some trace than the specification might (stably) refuse after the same trace.

3.1 A First Look at the Failures Normal Form

In the case that there is no unbounded internal progress (which would be divergence in the failures-divergences model), then each trace necessarily leads to a stable state and the normal form in the failures model can again be expressed as a minimal mutual recursion [1], shown below. Here the form of each clause is somewhat more complex than was the case for the traces normal form, reflecting the finer discrimination of the richer model.

$$\begin{aligned} P'(i) = (&\Box\ a \in A(i) \bullet a \to P'(after(i, a))) \\ &\triangleright \\ &(\sqcap\ m \in M(i) \bullet \Box\ a \in m \bullet a \to P'(after(i, a))) \end{aligned}$$

where $\triangleright$ is the *untimed timeout* operator[3], A and *after* play analogous roles to those in the traces normal form, and $M(i)$ is a (non-empty, finite) set of

[2] when we say 'failures' without the prefix 'stable-' it is shorthand for 'stable-failures'

[3] $P \triangleright Q$ may behave like P, but it will always offer the initial actions of Q. $P \triangleright Q = Q \ \Box\ P \sqcap Q$ (which is independent of bracketing).

incomparable subsets of $A(i)$, the *minimal acceptances* of $P'(i)$. (The restriction that $M(i)$ is a non-empty set of minimal acceptances is dropped in Section 4.) While $A(i)$ represents the choice of events immediately possible for $P'(i)$, the set $M(i)$ encapsulates the range of nondeterminism within that choice. Each of the elements of $M(i)$ is a set of events that $P'(i)$ may nondeterministically choose to offer; it is not generally possible for an external observer to tell which minimal acceptance has been chosen by $P'(i)$, but one can rely on all events from at least one minimal acceptance being offered by the process. This fact is crucial in the construction of WDF_S below.

Note that a process $P'(i)$ that may immediately deadlock has $M(i) = \{\emptyset\}$, and the failures normal form of a deterministic process has $M(i) = \{A(i)\}$ for each i. A process $DF(X)$ – DF here stands for 'deadlock free' – that can do any sequence of events drawn from X and is the most nondeterministic such process that can never deadlock, has the single-state normal form with $A(i_0) = X$ and $M(i_0) = \{\{x\} \mid x \in X\}$ and singleton index set $\{i_0\}$.

3.2 The Restricted Failures Watchdog Assertion

In contrast to the traces case, the stable-failures watchdog assertion is a deadlock freedom check (in the stable-failures model).

Given a watchdog process $WDF_S(i_0)$, the assertion $S \sqsubseteq_{\mathcal{F}} I$ is transformed to the watchdog assertion

$$I \underset{\alpha(S,I)}{\parallel} WDF_S(i_0) \quad \text{deadlock free } [\mathcal{F}] \tag{2}$$

which asserts that the parallel composition of I and the watchdog process, synchronised on all events of the original assertion, is deadlock free in the failures semantics. (The test for deadlock freedom of a process P can be expressed as a failures refinement check against the process $DF(\alpha(P))$, but such checks are sufficiently common to have a special form of assertion in the FDR meta-language.)

3.3 The Restricted Failures Watchdog Process

A suitable failures watchdog process is $WDF_S(i_0)$ where WDF_S is defined (relative to the stable-failures normal form of S) in $\mathrm{CSP_M}$ as follows:

```
1: channel trace_error_, spec_stopped_
2: sigma = Events − {trace_error_, spec_stopped_}
3:
4: WDF_S(i) =
5:     (□ a : A(i) • a → WDF_S(after(i, a)))
6:     □
7:     (□ a : sigma − A(i) • a → trace_error_ → STOP)
8:     ▷
9:     N(i) == 1 and empty(M_{i,1}) &
10:        – – spec state can stop (so has one minimal acceptance)
```

11: $\quad spec\_stopped\_ \rightarrow WDF_S(i)$
12: $\quad \Box$
13: $\quad N(i) > 0$ and not $empty(M_{i,1})$ &
14: $\quad\quad$ – – spec state cannot stop (and has at least 1 min acceptance)
15: $\quad\quad (\sqcap\ Y \in \{\{m_1, ..., m_{N(i)}\} \mid m_1 \in M_{i,1}, ..., m_{N(i)} \in M_{i,N(i)}\}$
16: $\quad\quad\quad \bullet \Box\ a \in Y \bullet a \rightarrow WDF_S(after(i, a)))$

We have written $N(i)$ above as shorthand for $card(M(i))$, so we have that $M(i) = \{M_{i,1}, ..., M_{i,N(i)}\}$. Line 1 defines two distinguished events *trace_error_* and *spec_stopped_* (identifiers ending in an underscore are conventionally reserved for machine-generated text). Line 2 defines *sigma* to be the set of all events *Events* except these distinguished events. So *sigma* contains (at least) all the events in $\alpha(S)$ and $\alpha(I)$.

3.4 Correctness

Recall that a refinement $P \sqsubseteq_{\mathcal{F}} Q$ holds in the stable-failures model if $traces(Q) \subseteq traces(P)$ and $failures(Q) \subseteq failures(P)$, where the latter are the sets of all possible observations (s, X) of a trace s of Q (or P) leading to a stable state that can refuse to engage in any event in X. Suppose we have normalised P to a mutual recursion of the form given in Section 3.1, and that i_s is the (unique) index such that $P'(i_s) = P/s$. Then a simple unwinding argument reduces the refinement check to a check that, for each trace s that both P and Q can perform, any operational state Q' that Q can reach on s (and so such that $Q/s \sqsubseteq_{\mathcal{F}} Q'$) has the following two properties:

1. $inits(Q') \subseteq A(i_s)$; and
2. if Q' is stable and can refuse X, then $\exists\, m \in M(i_s)$ such that $m \cap X = \emptyset$.

The first property says that Q' cannot offer more than P/s might, and the second says that Q' must fulfil at least one of the promises that P/s makes about what is offered.

We need to show that $S \sqsubseteq_{\mathcal{F}} I$ iff $I \underset{\alpha(S,I)}{\parallel} WDF_S(i_0)$ is deadlock free in $\mathcal{F}$. We argue both contrapositives.

Suppose that $S \sqsubseteq_{\mathcal{F}} I$ does not hold; then there are two generic possibilities, which we narrow by considering minimal counterexamples: either there is a trace $s^\frown\langle a\rangle \in traces(I)$ such that $s \in traces(S) \cap traces(I)$ but $s^\frown\langle a\rangle \notin traces(S)$; or there is a failure $(s, X) \in failures(I) - failures(S)$ where $s \in traces(S)$.

In the first case, since $s^\frown\langle a\rangle$ is a trace of I, it is possible for the left-hand-side of the parallel composition to perform s and reach a state that can do a; and, as s is a trace of S, a possible execution of the right-hand-side is always to take line 5 of the definition of WDF_S. So we reach a position where $i = i_s$ is by assumption such that $a \notin A(i)$, and so line 7 can contribute a. Thus the parallel composition can evolve by synchronising on a, after which the watchdog can do *trace_error_* and prevent I doing any further events, by becoming $STOP$. So

the system can deadlock on the trace $s^\frown\langle a, \mathit{trace\_error\_}\rangle$ and is therefore not deadlock free.

In the second case, again we can reach a state where the system has performed s and I has reached a state that can stably refuse X. The watchdog can then 'timeout' to the choice at lines 9–16. The boolean guard at line 9 cannot be true (as then S/s can refuse anything, in particular X), but the guard at line 13 is true, so the watchdog reduces to the last two lines. Property 2 above is not satisfied, since the refinement doesn't hold, and therefore for each $M_k \in M(i_s)$ there is a witness $m_k \in M_k \cap X$. Then $\{m_0, ..., m_{N(i_s)}\}$ is one of the possible nondeterministic choices for Y at line 15, and so the watchdog can offer only that Y (and we are only interested in the possibility of deadlock), while the left-hand-side of the parallel can refuse it (because $Y \subseteq X$, and refusals are closed under subset). So, again, the composition is not deadlock free.

Conversely, suppose the implementation in parallel with the watchdog deadlocks after some trace s_i, say; let us consider the state of the watchdog at that point. One possibility is that the watchdog is on line 7; but it cannot be before the *trace_error_* event, since that can happen without the cooperation of its peer, and in any case the timeout operator could make internal progress and transfer control to the lower half of the process. So in this case it must have already done the *trace_error_* event; but that can only happen when I/s_i can do an event that S/s_i cannot, and so the refinement cannot hold. Similarly, if the guard at line 9 is true, the *spec_stopped_* event can happen autonomously, and repeatedly, and no deadlock is possible (this is, of course, the reason for the inclusion of this clause: if the specification is allowed to deadlock after some trace, we must ensure that this new composition does not deadlock at that point). So the guard at line 13 must be true, and it will have picked a particular Y with an element of each element of $M(i)$. Since there is a deadlock after trace s_i, I/s_i must be able to refuse all of Y, which implies that some operational state Q', reachable by I on s_i, refuses all of Y. But then I/s_i has a refusal X $(= Y)$ that does not satisfy property 2 above, and the refinement does not hold (as (s_i, X) is a failure of I but not S). This completes the proof.

In the stable-failures model, $P \setminus X$ is deadlock free iff P is deadlock free, for arbitrary process P and set of events X. So this transformed assertion is suitable for our overall game-plan.

4 General Watchdog Transformations for the Failures Semantics

We have explicitly assumed, in the previous section, that $M(i)$ is non-empty for all i, and we have implicitly made use of an analogous assumption about I, although it may not be immediately obvious where. But these assumptions do not hold, in general, for processes in the stable-failures model. The reason it is the *stable*-failures model is that we only record failures that are stable. The definition of refinement mentions traces as well as failures because a process

might be able to perform some particular trace but never reach a stable state without subsequently doing a further event; and possibly not even then.

Such instability is analogous to divergence in the failures-divergences model, but the two denotational semantics (and, consequently, the two algebras) treat it rather differently. In particular, there are many different unstable processes, in contrast to the single and catastrophic divergence: in fact, any subsequent behaviour is possible in the stable-failures model. If the definition of distributed nondeterministic choice is extended so that (the usually illegal term) $\sqcap\, \emptyset$ is identified with the pure unstable process $\mathsf{div} = P \setminus \{a\}$, where $P = a \rightarrow P$, then the definition of failures normal form in Section 3.1 continues to make sense even when some $M(i)$ are empty. Note that div, and more generally $\mathsf{div} \,\Box\, Q$ for any Q such that $\mathit{inits}(Q) \neq \mathit{Events}$, are rather miraculous in the stable-failures model: there are events that they can never do, but equally never refuse to do; this contradicts one of the (quite intuitive) axioms of the failures-divergences model. Also, not every divergence corresponds to an instability: because div is a unit of $\sqcap$, it is only when an operational divergence is unavoidable that it gives rise to instability; there are no refusals belonging to a trace only when there are no finite maximal τ-chains from any state that can be reached on that trace.

4.1 The General Failures Watchdog Process

The previously vacuous conjunct $N(i) > 0$ in the guard at line 13 now comes into play: if S/s is denotationally unstable (i.e., it has no minimal acceptances) then neither of the guards in the second half of the definition of $WDF_S(i_s)$ is true, and therefore the body of the definition degenerates to $\ldots \triangleright STOP$. Since the alphabet of I is a subset of the synchronisation set $\alpha(S, I)$, the parallel composition must be able to deadlock... unless I/s itself is unstable. (The semantics of the parallel operator essentially make the refusals of the compound the pairwise unions of refusals of the components, and so if either side has none, then so does the whole.) And $\mathsf{div} \,\Box\, \ldots$ has no refusals on the empty trace. But this is precisely what we want: if S has no refusals on s, we want I to have none, also, as otherwise that would be a failure too many.

Thus the existing definition of WDF_S will serve admirably in the general case.

4.2 The General Failures Watchdog Assertion

Unfortunately, the same feature of the semantics that gives us the correct behaviour when the specification is unstable, allows the possibility of masking a trace error if the implementation can become unstable after performing some illegal event: the *trace_error_* event may happen, since the implementation cannot influence that, and thereafter no event in $\alpha(S, I)$ can happen. But, equally, they may not be refused by a miraculous state of the implementation: $(\mathsf{div} \,\Box\, \ldots)[|\ \ldots\ |]STOP$ is deadlock free, provided the left-hand-side cannot make a transition to a stable state.

The full-abstraction results in [1] establish that we must be able to separate the test into one on traces, and another for immediate deadlock; but this would require two traversals of the complete state-space of I and, worse, two different transformations of S, including normalisation in two different models. We much prefer to find a modification to the transformed assertion that allows the test to be completed in a single check.

In fact, the change required is quite straightforward: we simply check

$$(I \triangle \mathit{trace\_error\_} \rightarrow \mathit{STOP}) \underset{\alpha(S,I)\cup\{\mathit{trace\_error\_}\}}{\parallel} \mathit{WDF}_S(i_0) \quad \text{deadlock free } [\mathcal{F}] \qquad (3)$$

where $\triangle$ is the CSP *interrupt* operator, which effectively adds a deterministic choice of doing its second argument to every state of its first; it is not compositionally definable in terms of the other operators of the language, and we believe that this conflation of the checks into a single check could not be encoded without it.

4.3 Correctness

Essentially, the argument of Section 3.4 carries through unchanged, apart from the claim that the *STOP* after the *trace_error_* event at line 7 introduces deadlock. This is not necessarily true in the presence of instability, as pointed out above.

Now, however, the *trace_error_* event cannot occur without the cooperation of the left-hand-side of the parallel; and since $\mathit{trace\_error\_} \notin \alpha(I)$, it must be the right-hand operand of the interrupt that does it. The left-hand-side of the parallel then becomes *STOP*, which (stably) deadlocks. So if *trace_error_* does occur, then both sides of the parallel are stable and deadlocked, so the whole parallel is also. It may be that I can make infinite internal progress instead of performing that event, but the interrupt operator ensures that it is always available, and the deadlock check will explore every possible execution, including those where it is eventually chosen.

5 Compression Strategies

As explained in Section 1, our motivation for the watchdog transformations presented above was the desire to improve the effectiveness of compressions. In this section we outline the approach we have taken to developing compression strategies that (attempt to) take full advantage of the increased amount of hiding that the watchdog transformations allow. This section is not intended to provide a full and final description of the compression strategies, which are still being developed. Our intention here is to outline our approach in order to indicate how the watchdog transformations above can be exploited.

The original refinement assertion has now been transformed to a suitable watchdog assertion. In the traces case, the watchdog assertion is a traces refinement, and in the failures case it is a failures deadlock-freedom assertion (which

can be expressed as a failures refinement). In both cases all events of the original assertion can be hidden without changing its truth or falsity, and we can construct an original counterexample from any watchdog counterexample.

Unfortunately, the naive approach of hiding all possible events and then compressing the whole process in one step is inefficient: FDR will traverse the full state space when calculating the compressed state machine.

Compression strategies generate a compressed state machine representation of a process. To explain how our compression strategies work, we begin by making four observations:

1. Compressing component processes before composing them can avoid the construction of large state machines that are later compressed.
2. Pushing hiding down through a process operator allows the composition to be compressed more effectively. Of course, events on which component processes synchronise with other processes cannot be hidden inside a parallel composition.
3. Rearranging the syntax tree of a process expression sometimes allows more hiding to be pushed down through process operators.
4. The syntax tree can be conveniently rearranged, without affecting semantics, when process operators are associative (perhaps allowing synchronization alphabets to change) and commutative.

These observations (in reverse order) motivate four principal compression activities: transform some or all parallel compositions to alphabetised parallel form, rearrange the order of alphabetised parallel compositions, push hiding down the syntax tree and, finally, apply one or more of FDR's compression functions at some places in the syntax tree. These activities are described in more detail in the following subsections.

5.1 Transforming parallel compositions to alphabetised form

$\mathrm{CSP_M}$ includes four parallel operators:

alphabetised parallel	$P \ {}_X\|\|_Y \ Q$
shared parallel	$P \ \|\|_X \ Q$
interleaving	$P \ ||| \ Q$
linked parallel	$P \ [a_1 \leftrightarrow b_1, ..., a_n \leftrightarrow b_n] \ Q$

The alphabetised parallel operator synchronises processes on specified alphabets and constrains them to perform events within these alphabets. The shared parallel operator synchronises processes on the specified alphabet and interleaves them on other events. Linked parallel synchronises events of one process with corresponding (linked) events of the other; events that are not thus linked can occur independently of the other process.

Alphabetised parallel is the only associative parallel operator. We prefer to describe it as pseudo-associative, since there is an obligation to manage the synchronisation alphabets. By saying that alphabetised parallel is pseudo-associative we mean:

$$P \ {}_X\|_{Y \cup Z} \ (Q \ {}_Y\|_Z \ R) = (P \ {}_X\|_Y \ Q) \ {}_{X \cup Y}\|_Z \ R.$$

We wish to express all parallel compositions in terms of the alphabetised parallel operator.

Interleaving and linked parallel can be represented in terms of shared parallel, renaming and hiding:

$$P \ ||| \ Q = P \underset{\emptyset}{\|} Q$$

and

$$P[a_1 \leftrightarrow b_1, ..., a_n \leftrightarrow b_n]Q = (rP \underset{\{c_1, ..., c_n\}}{\|} rQ) \setminus \{c_1, ..., c_n\}$$

where $rP = P \ [[a_1 \leftarrow c_1, ..., a_n \leftarrow c_n]]$, $rQ = Q \ [[b_1 \leftarrow c_1, ..., b_n \leftarrow c_n]]$ and $c_1, ..., c_n$ are distinct new events.

To convert shared parallel into alphabetised parallel, we need two renamings:

$$P \underset{X}{\|} Q = (rP \ {}_{X \cup \alpha(rP)}\|_{X \cup \alpha(Q)} \ Q) \ [[b_1 \leftarrow a_1, ..., b_n \leftarrow a_n]]$$

where $rP = P[[a_1 \leftarrow b_1, ..., a_n \leftarrow b_n]]$, $a_1, ..., a_n$ are the events outside X that are in both $\alpha(P)$ and $\alpha(Q)$, and $b_1, ..., b_n$ are distinct new events. Essentially, if we simply put two processes P and Q from a shared parallel composition into alphabetised parallel over their respective alphabets, then synchronisation would occur on any event that both P and Q could perform. We want synchronisation to occur on only the set X (the set over which the processes synchronise in the shared parallel composition). So, before composing we rename those events of P on which undesired synchronisation would otherwise occur, and rename them back afterwards.

5.2 Reordering alphabetised parallel compositions

We are working on several alternative heuristics for the reordering of parallel process composition. Space does not allow sufficiently detailed description of these heuristics, so we outline the approach here and discuss only the simplest heuristic in any detail.

Recall that we are trying to gain some advantage by pushing hiding down a syntax tree towards the leaves. In the traces case, the tree will initially look something like Figure 1, which depicts the case $I = Impl1 \ {}_{\alpha(Impl1)}\|_{\alpha(Impl2)} \ Impl2$.

Unfortunately, the watchdog process generated by transforming the specification synchronises with the implementation on all the events, $\alpha(S, I)$, of the original assertion, so hiding cannot be pushed far in. Therefore we want to transform

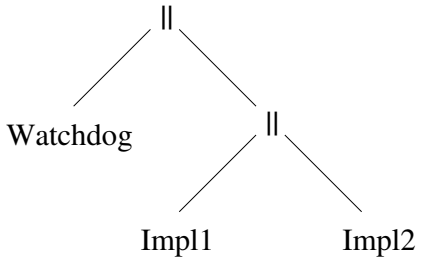

Fig. 1. Watchdog process at the top of the syntax tree.

the syntax tree, moving the watchdog process downwards; we need associativity and commutativity to do this.

Our simplest heuristic pushes the watchdog process as low in the syntax tree as possible. To illustrate this, consider again the syntax tree shown in Figure 1. Now, neither the implementation process nor any of its component processes can hide events that are (or are eventually renamed to) events in the alphabet of the implementation process. As already noted, this places a severe restriction on the effectiveness of compressions applied below this point in the syntax tree.

To make it possible to hide more events low in the syntax tree we can change the tree in Figure 1 to the one shown in Figure 2.

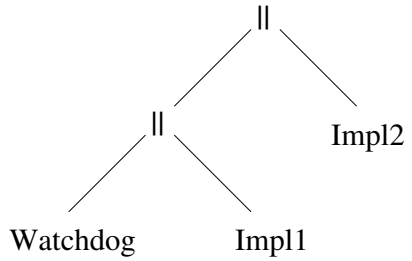

Fig. 2. Transformed syntax tree with the watchdog process moved down

That is, we can change

$$Watchdog \ {}_{sigma}\|_{\alpha(Impl1,Impl2)} \ (Impl1 \ {}_{\alpha(Impl1)}\|_{\alpha(Impl2)} \ Impl2)$$

to

$$(Watchdog \ {}_{sigma}\|_{\alpha(Impl1)} \ Impl1) \ \ {}_{sigma}\|_{\alpha(Impl2)} \ Impl2$$

and so allow events on which $Impl2$ does not synchronise to be hidden and then compressed from the composition of the watchdog process and $Impl1$. Once

the watchdog process has been moved down the syntax tree, there is scope to re-arrange the components of the implementation, depending perhaps on the respective sizes of their interface alphabets. In the case described above, it might be worth composing *Impl*2 with the watchdog and then composing the result with *Impl*1.

Clearly, the reordering heuristic described above is very simple, and we do not claim it is optimal. Nevertheless, it has the virtue of addressing an important limitation - outlined above - on the effectiveness of compressions: that one cannot 'compress away' transitions of a state machine that are needed for synchronisation. The framework we have developed allows us to experiment conveniently with a variety of heuristics and so compare them empirically; the emerging results, however, are postponed to a follow-up paper due to lack of space.

5.3 Pushing hiding down the syntax tree

It is easy to justify pushing hiding down any particular syntax tree. Once we have a syntax tree that composes processes in an optimal order, we are ready to push hiding down this tree, through the process operators.

Hiding can be pushed through alphabetised parallel compositions using the law:

$$(P \ {}_X\|_Y \ Q) \setminus H = (P \setminus H \cap X{-}Y \ {}_X\|_Y \ Q \setminus H \cap Y{-}X) \setminus H \cap X \cap Y$$

which exploits the fact that $\cap$ and $-$ associate. A corresponding law for shared parallel is:

$$(P \underset{X}{\|} Q) \setminus H = (P \setminus H{-}X \underset{X}{\|} Q \setminus H{-}X) \setminus H \cap X$$

These laws state that we can hide events from the set H at the level of P and Q, gaining efficiency, but only if P and Q cannot synchronise on these events when composed. There are similar laws, not presented here, that allow hiding to be pushed through other $\mathrm{CSP_M}$ operators.

5.4 Applying compressions

There is some choice of which of FDR's compression functions to apply to which compositions (i.e., at which nodes of the syntax tree). For example, one might decide to apply a given compression operator at all leaf nodes, and another at all interior nodes. It is important not to compress too high up in the syntax tree because the full state space – traversed by FDR's compression functions – may become prohibitively large. There is considerable scope to choose the compression operator (if any) to apply at a node based on the nature of the composition and the processes that are being composed. We are currently experimenting with such heuristics.

Our simplest compression strategies are parameterised by a single compression function, which is applied at each interior node of the transformed syntax tree.

6 Implementation

Recall that normal form processes have their initials sets explicitly available. So, the watchdog process transformation can be performed in two parts: first normalise, then transform the normal-form process. FDR already has an efficient implementation of normalisation, and we make use of this to obtain the initials after any trace.

Before performing a refinement check, FDR *normalises* (the state machine for) the specification process – it transforms the specification state machine into a form where there is a unique operational state for each visible trace of the specification process. This operational normal form corresponds directly to the algebraic normal forms. Further, using the scripting interface to FDR it is possible to expose the normal-form of a specification and recurse over it to construct a state machine for the watchdog process. In this way, we have implemented the transforms WDT_S and WDF_S. Given the watchdog process, it is then straightforward to generate the watchdog assertion, and then hide the events of the original assertion (all at the level of state machines).

We have implemented a framework that interfaces with the FDR compiler and its normalisation functionality. This framework performs the watchdog transformations described in this paper and allows us to experiment with a range of compression strategies. The fruits of this experimentation will be reported in due course.

6.1 Complexity

A disappointing worst-case bound can be deduced for this strategy from Valmari and Kervinen's result [15] on the logarithmic complexity of refinement checking when the specification is a simple composition using alphabetised parallel, compared to when arbitrary parallel operators are allowed. Transforming general parallel compositions to alphabetised parallel must be EXPSPACE-hard, or be capable of producing output exponentially larger than its input, as otherwise one could transform any specification into that form and so reduce an EXPSPACE-complete problem to an NPSPACE one.

In practice, the pathological worst cases seem to arise infrequently: the normalisation procedure produces a state machine potentially exponentially larger than its argument, but it usually leaves it roughly the same size. It actually makes it significantly smaller often enough that it is a popular choice of compression function.

Here, the blow-up appears to arise mainly in the numerous large renamings that are required to implement nonsynchronising parallels in terms of the alphabetised form. One essentially needs to invent a new name for each way in which an event can arise at the top of the composition, from different combinations of leaf processes and pre-renaming events. But this cost is already paid once in the FDR "supercompiler" data structures, where it has rarely proven prohibitive. So there is hope that a fairly efficient coding of the transformation will perform satisfactorily in the majority of cases.

7 Related work

The third author originally proposed the general approach in the course of discussion with Jay Yantchev.

Atanas Parashkevov, a student of Yantchev's, has recently and independently developed that original idea in a different direction, with a view to exploiting it for BDD-based tools. The intention is to hide events to improve the performance of BDD algorithms (rather than to improve the performance of compression algorithms). To this end, Parashkevov has formulated, though not yet published, an Observer process for the traces semantics; this is essentially the simple watchdog process in [2].

8 Acknowledgements

The third author wishes to thank Jay Yantchev for their conversations on these ideas in 1997. The second and fourth authors were partly funded by the EU Framework V project DSoS: Dependable Systems of Systems.

References

[1] Roscoe, A.W.: The Theory and Practice of Concurrency, Prentice Hall (1998).

[2] Zakiuddin, I., Moffat, N., Goldsmith, M., Whitworth, T.: Property Based Compression Strategies. In: *Proceedings of Second Workshop on Automated Verification of Critical Systems* (AVoCS 2002), University of Birmingham, 15-16 April 2002.

[3] de Roever, W. P., de Boer, F., Hannemann, U., Hooman, J., Lakhnech, Y., Poel, M., Zwiers, J.: Concurrency Verification: Introduction to Compositional and Noncompositional Proof Methods. Cambridge Tracts in Theoretical Computer Science, 54. (2001)

[4] Clarke, E.M., Grumberg, O.: Model Checking and Abstraction, In: *ACM Transactions on Programming Languages and Systems*, ACM Press (1992) 1512-1542

[5] Clarke, E., Filkorn, T., Jha, S: Exploiting symmetry in temporal logic model checking, In: *Proceedings of 5th International Conference on Computer Aided Verification*, (1993)

[6] Ip, C.N., Dill, D.L.: Better Verification Through Symmetry. Computer Hardware Description Languages and their Applications. Elsevier Science Publishers B.V., Amsterdam, Netherland (1996)

[7] Emerson, E.A., Sistla, A.P.: Symmetry and Model Checking. In: *Formal Methods in System Design: An International Journal*, Kluwer Academic Publishers (1994) 105-131

[8] Valmari, A.: A stubborn attack on state explosion, 2nd Workshop on Computer Aided Verification, New Brunswick, NJ, Lecture Notes in Computer Science 531, Springer-Verlag (1987) 156-165

[9] Peled, D., Pnueli, A.: Proving partial order properties, *Theoretical Computer Science*, 126. (1994) 143-182

[10] Godefroid, P.: Partial-order Methods for the Verification of Concurrent Systems, Springer-Verlag Berlin and Heidelberg (1996)

[11] McMillan, K.L.: Symbolic Model Checking. Kluver Academic Press. (1993)
[12] Formal Systems (Europe) Ltd.: FDR User Manual, 1992-99.
[13] Roscoe, A.W., Goldsmith, M., Gardiner, P.H.B., Jackson, D., Scattergood, B., Hulance, J.: Hierarchical Compression for Model-Checking CSP, or How to Check 10^{20} Dining Philosophers for Deadlock. In: *Proceedings of 1st TACAS*, BRICS Notes Series NS-95-1, Department of Computer Science, University of Aarhus, 1995; also Springer LNCS 1019.
[14] Scattergood, J.B.: Tools for CSP and Timed CSP. Oxford University D.Phil. thesis, 1998.
[15] Valmari, A., Kervinen, A.: Alphabet-Based Synchronisation is Exponentially Cheaper. In: Brim, L., Jancar, P., Kretinsky, M., Kucera, A. (eds.): *CONCUR 2002 – Concurrency Theory, 13th International Conference*, Brno, Czech Republic, August 20-23, 2002, Proceedings. Lecture Notes in Computer Science, Vol. 2421. Springer 2002.