DOI 10.1007/s00165-007-0065-0
BCS © 2007
Formal Aspects of Computing (2008) 20: 277–294

Formal Aspects of Computing

Specification of communicating processes: temporal logic versus refusals-based refinement

Gavin Lowe

Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford, OX1 3QD, UK.
E-mail: gavin.lowe@comlab.ox.ac.uk

Abstract. In this paper we consider the relationship between refinement-oriented specification and specifications using a temporal logic. We investigate the extent to which one can check whether a program in a process algebra, such as Communicating Sequential Processes (CSP), satisfies a temporal logic specification using a refinement-based model checker, such as FDR. We consider what atomic formulae are appropriate in a temporal logic for specifying communicating processes, in particular where one wants to talk about the availability of events. We then show that, perhaps surprisingly, the standard stable failures model is not adequate for capturing specifications in such a logic: instead the refusal traces model must be used. We formalise the logic by giving it a semantics in this model. We show that the temporal operators *eventually* and *until*, and negation, cannot, in general, be tested for via simple refinement checks. For the remaining fragment of the logic, we present a translation into simple refinement checks. Finally, we show that refusal traces equivalence is characterised by a slightly augmented version of that fragment.

Keywords: Temporal logic; Specification; Refinement testing; Refusals; CSP

1. Introduction

Model checking is a powerful tool for the analysis of systems. A finite-state model of a system is checked against a specification. How should that specification be captured? Two important approaches are as follows.

Temporal logic The specification is captured in a temporal logic, such as LTL [Pnu81], with operators to capture temporal properties such as "always" and "until". We discuss temporal logics in more detail below. A model checker is used to test directly whether the system satisfies the specification. Advocates of this approach claim that writing specifications in this form is more straightforward than other approaches. This approach is epitomised by the SPIN tool [Hol97, Hol03].

Refinement The specification is captured as a program, written in the same language as the system in question; the specification captures the range of allowable behaviours. A model checker is then used to test whether the behaviours of the system are a subset of those of the specification, i.e. whether the system *refines* the specification. Refinement is a pre-congruence, which means that this approach can be used in step-wise, compositional development. This approach is epitomised by FDR [Ros94], the model checker for Communicating Sequential Processes (CSP) [Hoa85, Ros97].

This paper aims to unite these two approaches. More precisely, we consider how communicating systems, modelled as CSP processes, can be checked against specifications expressed in a temporal logic, using a refinement-based model checker such as FDR.

Correspondence and offprint requests to: G. Lowe, E-mail: gavin.lowe@comlab.ox.ac.uk

Linear temporal logic formulae are built up using constructs such as the following:

Atomic predicates describing single states; typically, such atomic predicates are application-specific; we will consider what atomic predicates are appropriate for specifying communicating systems, as typically modelled using CSP.

Boolean connectives typically conjunction, disjunction and negation.

Temporal operators typically:

- $\bigcirc \phi$ ("next ϕ"): in the next state, ϕ will hold;
- $\Box \phi$ ("always ϕ"): in all subsequent states, ϕ will hold;
- $\Diamond \phi$ ("eventually ϕ"): in some subsequent state, ϕ will hold;
- $\phi \,\mathcal{U}\, \psi$ ("ϕ until ψ"): ϕ holds in every state until ψ becomes true; note that ψ must hold eventually;
- $\phi \,\mathcal{R}\, \psi$ ("ϕ releases ψ"): either ψ holds in every state, or ψ holds in every state up to and including a state in which ϕ is true; informally, ϕ releases the process from the obligation of satisfying ψ; note that ϕ need not ever hold.

Formulae in such a linear temporal logic talk about properties of single executions of a process. A process P satisfies such a formula ϕ if *every* execution of P satisfies ϕ; we write $P \models \phi$ in this case.

In this paper, we study the relationship between temporal logic and refinement-based specifications. In particular, we ask for which temporal logic formulae ϕ can one find a CSP specification process $Spec(\phi)$ such that

$$P \models \phi \ \ \textit{iff} \ \ Spec(\phi) \sqsubseteq P, \qquad \text{for all } P.$$

For simplicity, we restrict ourselves to divergence-free processes P (i.e. processes that cannot perform an unbounded amount of internal activity without communicating externally): a process that can diverge is normally, ipso facto, incorrect. However, we will sometimes allow the specification process $Spec(\phi)$ to be divergent. The refinement on the right (and whether P is divergence-free) can then be tested using a model checker such as FDR.

Note that we restrict ourselves to *simple* refinement checks, by which we mean checks where the left hand side is independent of the process P in question, and the right hand side is just P. The consideration of more general refinement checks is left for future work.

We consider only linear-time, as opposed to branching-time, temporal logic. Branching-time temporal logics, such as CTL [CES86], allow one to specify the *existence* of a behaviour with a certain property; this seems less useful than specifying that *all* behaviours satisfy that property; further, such properties are not refinement-closed, which is clearly a necessary condition to make simple model checking possible.

It is easy to see that refinement checking can capture properties that can not be captured using linear temporal logic. It is well known that one cannot capture the property "an a happens in every other state" (e.g. the second, fourth, sixth, etc., states) using LTL [Wol83]. However, it is trivial to capture this property as a refinement specification: P satisfies this property if and only if it refines the process that performs an a in the second, fourth, sixth, etc., states, but allows arbitrary events in the first, third, fifth, etc., states, namely the process $Spec \mathrel{\widehat{=}} \$x : \Sigma \to a \to Spec$.[1]

In the next section, we give a brief overview of the syntax and semantics of the fragment of CSP that we will be using in this paper. In Sect. 3, we consider temporal logic formulae in more detail. In Sect. 3.1 we consider which atomic formulae are useful for specifying communicating processes, such as those typically modelled in CSP. In Sect. 3.2 we show that the standard stable failures model of CSP is not suitable for such analyses: we show that the model does not give enough information about processes in order to give a natural semantics to some simple formulae. This was surprising to the author: the stable failures model is the standard model for capturing safety and liveness properties. Instead, the refusal traces (or "refusal testing") model must be used: we give an overview of this model in Sect. 3.3.

In Sect. 3.4, we give a formal semantics to the temporal logic in the (infinite) refusal traces model. We then show, in Sect. 3.5, that checking the satisfaction of such formulae is not always possible using simple, finite-state, refinement-based model checking, in particular if one includes the eventually ($\Diamond$) or until ($\mathcal{U}$) temporal operators, or negation.

[1] *Spec* nondeterministically chooses an event x from the set Σ of all events and offers it to the environment, i.e. it is willing to do any event initially; if x is performed, it then offers a; if the a is performed, it then recurses to its initial state.

In the remainder of the paper we consider the fragment of the logic excluding those operators, i.e. the bounded, positive fragment. In Sect. 4, we give a translation of formulae in this fragment into refinement-based checks, which can be analysed using a model checker such as FDR.

In Sect. 5 we consider the expressiveness of the bounded, positive fragment. We show that with a small addition, the refusal traces equivalence is characterised by this fragment. We consider this result to be of more theoretical than practical importance: the fact that the logic characterises the equivalence does not show that the logic is useful for specifying processes, nor vice versa; however, it does give some evidence that the model and logic are appropriate to one another. We sum up in Sect. 6.

2. Communicating Sequential Processes

Communicating Sequential Processes [Hoa85, Ros97] is a process algebra for describing programs or *processes* that interact with their environment by communication. In this section we give a brief overview of the syntax and semantics of CSP; for more details, see [Ros97].

CSP processes communicate via atomic events, from some set Σ; we restrict ourselves to finite alphabets in this paper.

The simplest process is $STOP$, which represents a deadlocked process that cannot communicate with its environment. The process div represents a divergent process, that performs an infinite amount of internal activity without ever communicating with its environment.

The process $a \to P$ offers its environment the event a; if the event is performed, it then acts like P. This is generalised by the process $?a : A \to P(a)$, which offers its environment the events from set A; if the event a is performed, the process then acts like $P(a)$.

The process $P \Box Q$ can act like either P or Q, the choice being made by the environment: the environment is offered the choice between the initial events of P and Q. By contrast, $P \sqcap Q$ may act like either P or Q, with the choice being made internally, and not under the control of the environment. For example, consider two vending machines, described as follows:

$$VM1 \mathrel{\widehat{=}} coin \to (tea \to VM1 \Box coffee \to VM1),$$
$$VM2 \mathrel{\widehat{=}} coin \to (tea \to VM2 \sqcap coffee \to VM2).$$

$VM1$, after insertion of a coin, offers the customer a choice between a tea and a coffee; $VM2$, on the other hand, chooses internally which drink to offer to the customer.

The process $\sqcap_{i \in I} P(i)$ represents an indexed nondeterministic choice between the processes $P(i)$ for $i \in I$; we restrict ourselves to finite I in this paper. The process $\$a : A \to P(a)$ represents a nondeterministic choice between the processes $a \to P(a)$ for $a \in A$. The process $P \rhd Q$ represents a sliding choice or timeout: the process initially acts like P, but if no event is performed then it can internally change state to act like Q. Prefixing binds tighter than each of the binary choice operators. The process $Chaos$ is the most nondeterministic, divergence-free process; it can be defined by $Chaos \mathrel{\widehat{=}} STOP \sqcap \$x : \Sigma \to Chaos$.

The process $P \underset{A}{\parallel} Q$ runs P and Q in parallel, synchronising on events from A. The process $P \setminus A$ acts like P, except the events from A are hidden, i.e. turned into internal, invisible events.

CSP can be given both an operational and denotational semantics; it is more common to use the denotational semantics when specifying or describing the behaviours of processes, although most tools act on the (congruent) operational semantics. Each of the denotational models represents a process by the set of behaviours it can exhibit: different models consider different types of behaviour, giving more or less information about the process.

A *trace* of a process is a sequence of (visible) events that a process can perform. The traces model represents a process P by the set of its traces, written $traces(P)$.

A *stable failure* of a process P is a pair (tr, X), which represents that P can perform the trace tr to reach a stable state (i.e. where no internal events are possible) where X can be refused, i.e. where none of the events of X is available. The stable failures model represents a process P by the set of its stable failures, written $failures(P)$. The stable failures model allows us to talk about which events are available—i.e. cannot be refused—whereas the traces model only allows us to talk about which events *might* be available.

Refinement is an important concept: $Spec$ is refined by P if all the behaviours of P are allowed by $Spec$:

$$Spec \sqsubseteq_T P \text{ iff } traces(Spec) \supseteq traces(P),$$
$$Spec \sqsubseteq_F P \text{ iff } failures(Spec) \supseteq failures(P).$$

FDR can be used to check these refinements. Note, though, that it considers only finite-state processes.

3. Temporal logic formulae: semantics and impossibility results

3.1. Atomic formulae

In this section we consider what are appropriate atomic formulae for specifying communicating systems. We clearly need to be able to talk about the performance of events. However, because we are dealing with communicating systems, where the performance of events depends upon the environment, we also want to be able to talk about the *availability* of events, in order to distinguish between nondeterministic and environmental choices; few existing temporal logics allow for the treatment of availability of events (the only example of which I am aware is [Jac92]). We therefore propose the following atomic formulae:

a (for a an event): the event a is available to be performed initially if the environment is willing to perform it, but no other event is performed.

$\mathsf{available}\, a$: the event a is available to be performed initially: if the environment were to attempt the event, it would be performed; however, this does not prevent a different event from being performed.

$\mathsf{deadlocked}$: the process is deadlocked; this is equivalent to $\bigwedge_{a \in \Sigma} \neg a$.

live: the process is live, i.e. not deadlocked; this is equivalent to $\bigvee_{a \in \Sigma} a$.

true, false: which have the normal meanings.

These atomic formulae are combined with the boolean and temporal operators described in the introduction.

Later we will define directly what it means for an individual behaviour to satisfy—or, more precisely, not contradict—a specification ϕ. We will say a process P *satisfies* ϕ—written $P \models \phi$—if all of the behaviours of P satisfy ϕ.

As an example to see why we want to be able to talk about the availability, rather than just the performance, of events, consider a vending machine that, after insertion of a coin, offers the customer the choice between a tea and a coffee. This requirement can be captured by

$$\phi \mathrel{\widehat{=}} \Box(coin \Rightarrow \bigcirc(\mathsf{available}\, tea \wedge \mathsf{available}\, coffee)),$$

where $a \Rightarrow \psi$ is shorthand for $\neg a \vee (a \wedge \psi)$. ϕ is satisfied by the process $VM1$ from Sect. 2, but not by $VM2$, which makes only one drink available. We need the $\mathsf{available}$ formulae in order to capture the shortcomings of the latter machine. Note that the use of the $\mathsf{available}$ formulae does not prevent other events from being performed; for example, ϕ is also satisfied by

$$VM3 \mathrel{\widehat{=}} coin \rightarrow (tea \rightarrow VM3 \;\Box\; coffee \rightarrow VM3 \;\Box\; chocolate \rightarrow VM3).$$

The following examples help to illustrate further the difference between the a and $\mathsf{available}\, a$ formulae (note that a implies $\mathsf{available}\, a$):

$$a \rightarrow Chaos \sqcap b \rightarrow Chaos \models a \vee b,$$
$$a \rightarrow Chaos \rhd b \rightarrow Chaos \models (a \wedge \mathsf{available}\, b) \vee b,$$
$$a \rightarrow Chaos \;\Box\; b \rightarrow Chaos \models (a \wedge \mathsf{available}\, b) \vee (b \wedge \mathsf{available}\, a).$$

In the remainder of this paper, we will consider the logic TL with the following grammar:

$$\begin{aligned}\phi \in TL ::= {} & \mathsf{true} \mid \mathsf{false} \mid a \mid \mathsf{available}\, a \mid \mathsf{live} \mid \mathsf{deadlocked} \mid \\ & \phi \wedge \phi \mid \phi \vee \phi \mid \neg\phi \mid \bigcirc\phi \mid \Box\phi \mid \phi \,\mathcal{R}\, \phi \mid \Diamond\phi \mid \phi \,\mathcal{U}\, \phi.\end{aligned}$$

3.2. Unsuitability of the stable failures model

In this section we show that the stable failures model [Ros97] is not sufficient to capture temporal logic specifications of the form described above, even for some rather simple specifications. More precisely, we present a temporal logic specification that when interpreted in the stable failures model allows arbitrary behaviours, contrary to its natural interpretation. I was surprised by this result: in my experience it is very rare not to be able to capture safety and liveness properties (for non-divergent processes) in the stable failures model.

Suppose we try to capture temporal logic specifications in the stable failures model. We could try to define a function $\mathcal{F}$ such that $\mathcal{F}[\![\phi]\!]$ gives the stable failures allowed by the formula ϕ. We would then have

$$\mathcal{F}[\![\mathsf{available}\,a]\!] = \{(\langle\rangle, X) \mid a \notin X\} \cup \{(tr, X) \mid tr \neq \langle\rangle \wedge X \in \mathbb{P}\,\Sigma\}.$$

The formula $\mathsf{available}\,a$ specifies that an a is available initially; therefore a may not be in the refusal set after the empty trace, but arbitrary refusals are allowed after non-empty traces. Further, we would have

$$\mathcal{F}[\![\bigcirc\,\mathsf{available}\,a]\!] = \{(\langle b\rangle, X) \mid b \in \Sigma \wedge a \notin X\} \cup \{(tr, X) \mid \mathrm{length}\,tr \neq 1 \wedge X \in \mathbb{P}\,\Sigma\}.$$

The formula $\bigcirc\,\mathsf{available}\,a$ specifies that an a is available after the first event; therefore a may not be in the refusal set after a singleton trace, but arbitrary refusals are allowed after all other traces. Disjunction naturally corresponds to union (as a consequence of the way we are defining satisfaction), so combining the previous two equations we get

$$\mathcal{F}[\![\mathsf{available}\,a \vee \bigcirc\,\mathsf{available}\,a]\!] = \mathcal{F}[\![\mathsf{available}\,a]\!] \cup \mathcal{F}[\![\bigcirc\,\mathsf{available}\,a]\!] = \{(tr, X) \mid tr \in \Sigma^* \wedge X \in \mathbb{P}\,\Sigma\}.$$

Note that the right hand side contains *all* stable failures. But this would mean that $\mathsf{available}\,a \vee \bigcirc\,\mathsf{available}\,a$ would be satisfied by *every* process, including processes such as $b \rightarrow STOP$ that intuitively do not satisfy this property.

The problem is that any behaviour with an empty trace satisfies $\bigcirc\,\mathsf{available}\,a$, because the behaviour has not yet progressed to the next state; whereas any behaviour with a non-empty trace satisfies $\mathsf{available}\,a$, because the behaviour does not record the refusal of a in the initial state.

The solution is to use the refusal traces model, which records refusal information throughout a trace, rather than just at the end.

3.3. The refusal traces model

The notion of refusal testing was introduced by Phillips in [Phi87]. The idea is to record refusal information throughout a trace. Refusal testing was adapted to CSP by Mukarram in [Muk93].

A *refusal trace* or *refusal test* is an alternating sequence of refusal information and events. More precisely, it takes one of two forms:

1. $\langle X_1, a_1, X_2, a_2, \ldots, X_n, a_n\rangle$, where each X_i is a refusal set, and each a_i is an event: this represents that the process can refuse X_1, perform a_1, refuse X_2, perform a_2, etc.
2. $\langle X_1, a_1, X_2, a_2, \ldots, X_n, a_n, \Sigma\rangle$: this is similar to the previous form, except finishes by refusing Σ, i.e. deadlocking.

(This is slightly different from the standard refusal traces model of [Muk93], where a refusal trace is of odd length, starting and finishing with a refusal set; our alternative form is equivalent, but will prove more convenient later).

Recall that refusal information is recorded only in *stable states*, where no internal activity is possible. Therefore, if an event is performed from an unstable state, no proper refusal can be observed before it. The refusal traces model uses a special null refusal value $\bullet$ to represent the absence of refusal information. This can occur because an event was performed from an unstable state, or because the observer made no attempt to observe refusal information. For example, the process $a \rightarrow STOP \rhd b \rightarrow STOP$ has refusal trace $\langle\bullet, a, \Sigma\rangle$: the a is performed from an unstable state, so no refusal information is associated with it; we use the $\bullet$ to indicate this. On the other hand, the process $a \rightarrow STOP \,\Box\, b \rightarrow STOP$ also has the refusal trace $\langle\{\}, a, \Sigma\rangle$, since the a is performed from a stable state (and, vacuously, no event from $\{\}$ is available). Finally, $a \rightarrow STOP \sqcap b \rightarrow STOP$ also has the refusal trace $\langle\{b\}, a, \Sigma\rangle$, since the a is performed from a stable state from which b is not available.

We therefore define a *refusal token* to be either a set of events or the null refusal: $RTOK \mathrel{\widehat{=}} \mathbb{P}\,\Sigma \cup \{\bullet\}$. We lift the normal set operations $\in$, $\notin$, $\cup$, $-$ and $\subseteq$ to operate over $RTOK$, treating $\bullet$ as being smaller than the empty set, so, for example, $x \notin \bullet$, $\bullet \subseteq \{\}$, $X \cup \bullet = X$.

If a process refuses X and performs a, we must have $a \notin X$. It is therefore convenient to define the one-step refusal traces by:

$$RT_1 \mathrel{\widehat{=}} \{\langle X, a\rangle \mid X \in RTOK \wedge a \in \Sigma - X\}.$$

We can then define PRT to be the partial refusal traces (i.e. those not ending in deadlock, corresponding to form 1 above), DRT to be deadlocked refusal traces (form 2 above), and RT to be all (finite) refusal traces:

$$PRT \mathrel{\widehat{=}} RT_1^*,$$

$$DRT \mathrel{\widehat{=}} \{tr^\frown\langle\Sigma\rangle \mid tr \in RT_1^*\},$$
$$RT \mathrel{\widehat{=}} PRT \cup DRT.$$

All processes satisfy certain healthiness conditions, and it is convenient to capture those conditions as axioms. We therefore define the refusal traces model to be those sets $R \subseteq RT$ satisfying the following conditions, where tr range over PRT, tr' ranges over RT, and A and B range over $RTOK$:

R1. $\langle\rangle \in R$;
R2. $tr^\frown tr' \in R \Rightarrow tr \in R$;
R3. $tr^\frown\langle A, a\rangle^\frown tr' \in R \wedge B \subseteq A \Rightarrow tr^\frown\langle B, a\rangle^\frown tr' \in R$;
R4. $tr^\frown\langle A, b\rangle^\frown tr' \in R \wedge A \neq \bullet \wedge tr^\frown\langle A, a\rangle \notin R \Rightarrow tr^\frown\langle A \cup \{a\}, b\rangle^\frown tr' \in R$.

R1 says that processes must at least be able to perform the empty trace. R2 says that the behaviours of processes are prefix-closed. R3 says that the behaviours of processes are closed under downwards closure of refusal tokens; note that this rule includes the case $B = \bullet$, and recall that $\bullet \subseteq A$ for all A. R4 says that if a process can stably refuse some set A, and cannot perform the event a, then that event can be added to the refusal set.

We write $\mathcal{R}[\![P]\!]$ for the refusal traces of process P. One can give semantic equations for all of the operators, as in [Muk93]. We give a few such equations below:

$$\begin{aligned}
\mathcal{R}[\![\mathsf{div}]\!] &= \{\langle\rangle\},\\
\mathcal{R}[\![STOP]\!] &= \{\langle\rangle, \langle\Sigma\rangle\},\\
\mathcal{R}[\![a \to P]\!] &= \{\langle\rangle\} \cup \{\langle X, a\rangle^\frown tr \mid a \notin X \wedge tr \in \mathcal{R}[\![P]\!]\},\\
\mathcal{R}[\![P \sqcap Q]\!] &= \mathcal{R}[\![P]\!] \cup \mathcal{R}[\![Q]\!],\\
\mathcal{R}[\![P \,\Box\, Q]\!] &= \{\langle\rangle\} \cup (\text{if } \langle\Sigma\rangle \in \mathcal{R}[\![P]\!] \cap \mathcal{R}[\![Q]\!] \text{ then } \{\langle\Sigma\rangle\} \text{ else } \{\}) \cup\\
&\quad \{\langle X, a\rangle^\frown tr \mid \langle X, a\rangle^\frown tr \in \mathcal{R}[\![P]\!] \wedge Q \;\mathsf{ref}\; X \vee \langle X, a\rangle^\frown tr \in \mathcal{R}[\![Q]\!] \wedge P \;\mathsf{ref}\; X\},
\end{aligned}$$

where $Q \;\mathsf{ref}\; X$ means that Q can refuse X initially:

$$Q \;\mathsf{ref}\; X \mathrel{\widehat{=}} \langle\Sigma\rangle \in \mathcal{R}[\![Q]\!] \vee (\exists a \cdot \langle X, a\rangle \in \mathcal{R}[\![Q]\!]) \vee X = \bullet.$$

Note that $P \,\Box\, Q$ can refuse X before the first event if both P and Q can; this is also true in the case $X = \bullet$.

Recall that we will be considering only divergence-free processes in this paper. Such processes have the property that all non-deadlocked behaviours can be extended, either with $\langle\Sigma\rangle$ (if the process is deadlocked) or some event a after a null refusal, $\langle\bullet, a\rangle$:

$$\forall tr \in \mathcal{R}[\![P]\!] \cap PRT \cdot tr^\frown\langle\Sigma\rangle \in \mathcal{R}[\![P]\!] \vee \exists a \cdot tr^\frown\langle\bullet, a\rangle \in \mathcal{R}[\![P]\!]. \tag{1}$$

Refinement in the refusal traces model is defined in the normal way:

$$Spec \sqsubseteq_R P \text{ iff } \mathcal{R}[\![Spec]\!] \supseteq \mathcal{R}[\![P]\!].$$

(We omit the subscript "R" on the refinement relation from now on). The model checker FDR was recently extended to include checking of refinement in this model. Note, in particular, that div is the top of the refinement relation: this refinement relation is therefore suitable only when the process on the right hand side is divergence-free.

In order to formally capture the semantics of certain temporal logic specifications, we will need to consider *infinite* refusal traces. We define the type of infinite refusal traces by

$$IRT \mathrel{\widehat{=}} RT_1^\omega.$$

The infinite refusal traces of a process can be calculated as the closure of the finite refusal traces[2]:

$$\mathcal{I}[\![P]\!] \mathrel{\widehat{=}} close(\mathcal{R}[\![P]\!]),$$

where

$$close(S) \mathrel{\widehat{=}} \{tr \in IRT \mid \forall tr' < tr \cdot tr' \in S\}.$$

[2] This follows from König's Lemma, and the fact that the corresponding operational semantics is finite-branching. It is possible to create a semantic model including infinite refusal traces where the finite-branching condition is relaxed: we do not do so here, for it would add unnecessary and distracting complexity, and take us outside of the space of processes checkable using FDR.

3.4. Formalising the logic

In this section we give a semantics to formulae of the logic. We will define a function $\mathcal{F}$ such that $\mathcal{F}[\![\phi]\!]$ gives the finite and infinite refusal traces allowed by ϕ. We can then define satisfaction of a formula ϕ by a process P by:

$$P \models \phi \Leftrightarrow \mathcal{R}[\![P]\!] \cup \mathcal{I}[\![P]\!] \subseteq \mathcal{F}[\![\phi]\!].$$

More precisely, $\mathcal{F}[\![\phi]\!]$ will include all traces that do not contradict ϕ, so will also contain behaviours where, informally speaking, ϕ has been neither demonstrated nor contradicted. For example:

- $\mathcal{F}[\![a]\!]$ will include $\langle\rangle$, for that is a trace of $a \rightarrow STOP$, which certainly satisfies a; the reason why the a has not been observed is that the observer did not attempt an a or did wait long enough;
- $\mathcal{F}[\![\bigcirc a]\!]$ will include $\langle \bullet, b\rangle$, for that is a trace of $b \rightarrow a \rightarrow STOP$, which satisfies $\bigcirc a$;
- $\mathcal{F}[\![\Diamond \phi]\!]$ will include *all* finite, non-deadlocked traces, for such traces can always be extended with a suffix that satisfies ϕ;
- $\mathcal{F}[\![\phi \mathcal{U} \psi]\!]$ will include all finite, non-deadlocked traces all of whose suffixes satisfy ϕ, even if ψ is never satisfied, since any such trace can be extended with a suffix that satisfies ψ.

In the following definitions b ranges over Σ, X ranges over $RTOK$, and tr ranges over $RT \cup IRT$. We write tr^i for refusal trace tr with the first i events and i refusals removed; we let i range over $0\,..\,\text{length}(tr)$ if tr is finite, or over all natural numbers if tr is infinite.

$$\begin{aligned}
\mathcal{F}[\![\mathsf{true}]\!] &\mathrel{\widehat{=}} RT \cup IRT,\\
\mathcal{F}[\![\mathsf{false}]\!] &\mathrel{\widehat{=}} \{\langle\rangle\},\\
\mathcal{F}[\![a]\!] &\mathrel{\widehat{=}} \{\langle\rangle\} \cup \{\langle X, a\rangle^\frown tr \mid a \notin X\},\\
\mathcal{F}[\![\mathsf{available}\, a]\!] &\mathrel{\widehat{=}} \{\langle\rangle\} \cup \{\langle X, b\rangle^\frown tr \mid a \notin X \wedge b \notin X\},\\
\mathcal{F}[\![\mathsf{live}]\!] &\mathrel{\widehat{=}} \{\langle\rangle\} \cup \{\langle X, b\rangle^\frown tr \mid b \notin X\},\\
\mathcal{F}[\![\mathsf{deadlocked}]\!] &\mathrel{\widehat{=}} \{\langle\rangle, \langle\Sigma\rangle\},\\
\mathcal{F}[\![\phi \wedge \psi]\!] &\mathrel{\widehat{=}} \mathcal{F}[\![\phi]\!] \cap \mathcal{F}[\![\psi]\!],\\
\mathcal{F}[\![\phi \vee \psi]\!] &\mathrel{\widehat{=}} \mathcal{F}[\![\phi]\!] \cup \mathcal{F}[\![\psi]\!],\\
\mathcal{F}[\![\neg\phi]\!] &\mathrel{\widehat{=}} RT \cup IRT - \mathcal{F}[\![\phi]\!] \cup \{\langle\rangle\},\\
\mathcal{F}[\![\bigcirc \phi]\!] &\mathrel{\widehat{=}} \{\langle\rangle, \langle\Sigma\rangle\} \cup \{\langle X, a\rangle^\frown tr \mid a \notin X \wedge tr \in \mathcal{F}[\![\phi]\!]\},\\
\mathcal{F}[\![\Box \phi]\!] &= \{tr \mid \forall i \bullet tr^i \in \mathcal{F}[\![\phi]\!]\},\\
\mathcal{F}[\![\phi\, \mathcal{R}\, \psi]\!] &\mathrel{\widehat{=}} \{tr \mid \forall i \bullet tr^i \in \mathcal{F}[\![\psi]\!] \vee \exists j < i \bullet tr^j \in \mathcal{F}[\![\phi]\!]\},\\
\mathcal{F}[\![\Diamond \phi]\!] &\mathrel{\widehat{=}} PRT \cup \{tr \mid tr \in IRT \cup DRT \wedge \exists i \bullet tr^i \in \mathcal{F}[\![\phi]\!]\},\\
\mathcal{F}[\![\phi\, \mathcal{U}\, \psi]\!] &\mathrel{\widehat{=}} \{tr \mid \exists i \bullet tr^i \in \mathcal{F}[\![\psi]\!] \wedge \forall j < i \bullet tr^j \in \mathcal{F}[\![\phi]\!]\} \cup\\
&\qquad \{tr \mid tr \in PRT \wedge \forall i \bullet tr^i \in \mathcal{F}[\![\phi]\!]\}.
\end{aligned}$$

Note that false allows the empty refusal trace, whereas one might have expected it to allow no refusal traces. This choice is necessary to allow several of the equivalences below. Note, however, that no divergence-free process can satisfy false, because of Eq. (1). We also always include the empty refusal trace in the semantics of $\neg\phi$, for similar reasons.

Note also that the formula available a specifies that an a is not refused initially, whereas the formula a additionally specifies that the first event (if any) is an a.

Note further that $\bigcirc \phi$ allows deadlock, so does not insist there is a next state: we interpret $\bigcirc \phi$ as meaning in *every* next state, ϕ holds; this is vacuously true if the process has no next state. The stronger requirement can be captured as live $\wedge \bigcirc \phi$.

We have chosen to give the semantics of formulae in terms of the denotational, as opposed to the operational semantics, as this is more convenient for the rest of the paper. A definition in terms of the operational semantics would have been equivalent, since the two semantics are congruent.

Returning to the example that motivated our use of refusal traces, we have

$$\begin{aligned}
\mathcal{F}[\![\mathsf{available}\, a \vee \bigcirc \mathsf{available}\, a]\!] = {}& \{\langle\rangle, \langle\Sigma\rangle\} \cup \{\langle X, b\rangle \mid b \notin X\} \cup \{\langle X, b, \Sigma\rangle \mid a \notin X \wedge b \notin X\} \cup\\
& \{\langle X, b, Y, c\rangle^\frown tr \mid (a \notin X \vee a \notin Y) \wedge b \notin X \wedge c \notin Y\}.
\end{aligned}$$

The interesting terms are the final two: if an a is in the initial refusal set, then a must not be in the second refusal set: this does not allow the trace $\langle\{a\}, b, \Sigma\rangle$ of $b \to STOP$, nor the trace $\langle\{a\}, b, \{a\}, c\rangle$ of $b \to c \to STOP$, for example.

The following lemma captures two basic properties of the semantics of formulae:

Lemma 1 The semantics of all formulae are subsets of $RT \cup IRT$ (so, in particular, no event performed is a member of the preceding refusal set), and satisfy healthiness condition R1 (so contain at least $\langle\rangle$).

We say that ϕ and ψ are equivalent if they have the same semantics:

$$\phi \equiv \psi \mathrel{\widehat{=}} \mathcal{F}[\![\phi]\!] = \mathcal{F}[\![\psi]\!].$$

The boolean operators satisfy the conditions for a boolean algebra. We state a few extra equivalences that will be useful later:

$$\begin{aligned}
\Box\,\phi &\equiv \mathsf{false}\,\mathcal{R}\,\phi,\\
\phi\,\mathcal{R}\,\psi &\equiv (\psi \wedge \phi) \vee (\psi \wedge \bigcirc(\phi\,\mathcal{R}\,\psi)),\\
a \wedge b &\equiv \mathsf{false}, \qquad \text{for } a \neq b,\\
\mathsf{live} &\equiv \neg\mathsf{deadlocked},\\
a \wedge \mathsf{deadlocked} &\equiv \mathsf{false},\\
\mathsf{available}\,a \wedge \mathsf{deadlocked} &\equiv \mathsf{false}.
\end{aligned}$$

For most of this paper we will restrict ourselves to the *negation restricted* fragment, denoted *NRTL*, by which we mean the fragment where all occurrences of the negation operator are applied to a formula of the form a (for $a \in \Sigma$), true, false, live or $\mathsf{deadlocked}$. In practice, we can assume that all negations are of the form $\neg a$, because of laws such as $\mathsf{true} \equiv \neg\mathsf{false}$ and $\mathsf{live} \equiv \neg\mathsf{deadlocked}$. The following lemma relates the semantics of formulae in this fragment to the healthiness conditions:

Lemma 2 The semantics of all formulae in *NRTL* satisfy healthiness conditions R2 and R3.

Condition R4 is not satisfied by all formulae. For example, consider the formula $\phi \mathrel{\widehat{=}} b \wedge \mathsf{available}\,a$. Then $\langle\{\}, b\rangle \in \mathcal{F}[\![\phi]\!]$ and $\langle\{\}, a\rangle \notin \mathcal{F}[\![\phi]\!]$, but $\langle\{a\}, b\rangle \notin \mathcal{F}[\![\phi]\!]$, contrary to R4.

3.5. Impossibility of checking *eventually*, *until* or negation

In this section, we show that we cannot check for properties involving the *eventually* ($\Diamond$), *until* ($\mathcal{U}$) or negation operators using *simple refinement testing*, by which we mean refinement checks where the right hand side is the process in question (we consider only refinement in the finite refusal trace model, as that is all that a model checker such as FDR can handle). We will show later that we are able to capture all formulae not using these operators.

Suppose, for a contradiction, that we could produce an appropriate specification process *Spec* for $\Diamond\,a$, i.e. such that $P \models \Diamond\,a$ iff $Spec \sqsubseteq_R P$, for all divergence-free P. Then *Spec* would have the refusal trace $\langle\Sigma - \{b\}, b\rangle^n \frown \langle\bullet, a\rangle$ for every n; i.e. it would allow an a after n bs, where only those bs were available. Now, the refusal traces of a process are prefix-closed (condition R2), so *Spec* would also have the refusal trace $\langle\Sigma - \{b\}, b\rangle^n$ for every n, and also all similar refusal traces with smaller refusals (condition R3). But that would mean that *Spec* would be refined by $P = b \to P$, which clearly does not satisfy $\Diamond\,a$. We deduce that we cannot, in general, deal with the eventually operator, $\Diamond$, using simple, finite trace refinement testing.

Note that $\Diamond\,\phi = \mathsf{true}\,\mathcal{U}\,\phi$, so we also cannot, in general, deal with the until ($\mathcal{U}$) operator.

These results are perhaps not surprising: most of an *eventually* or *until* property is captured by the infinite traces, and this is not reflected in a finite-state check. This can be seen as a consequence of the result in [Ros05] that only predicates that are closed in the standard topology on the failures divergences model [Ros97] can be expressed using simple refinement tests: properties that require one to examine the infinite behaviours are not closed.

I do not consider the inability to deal with these operators to be a major shortcoming: I have never encountered a situation where an unbounded eventually specification was appropriate, as opposed to a bounded specification, such as "ϕ becomes true within the next n steps";[3] where the appropriate value for n is not known, it can be found by trial and error.

[3] Such properties can be checked with FDR, for n up to a few thousand; note that there is no need to consider n greater than the diameter of the process graph, which is typically much smaller.

Note also that $\Diamond\,\phi = \neg(\Box\,\neg\phi)$. It turns out that we are able to deal with the always ($\Box$) operator; hence we cannot, in general, deal with the negation ($\neg$) operator. We can, however, deal with negation as it is used in the negation restricted fragment, defined above. Negation does not seem to be useful outside of this fragment. For example, the formula $\neg\,\textsf{available}\,b$ specifies that b must be included in every initial refusal set, so does not allow the empty or null refusals initially; hence, by healthiness condition R3, it allows only the empty and deadlocked traces, and so is satisfied only by $STOP$. We do not consider $\neg\,\textsf{available}\,b$ to be a particularly useful form of specification: the specification $\neg b$ is more often applicable.

3.6. The bounded, positive fragment

In the remainder of this paper we consider the bounded, positive fragment $BPTL$ of TL, i.e. excluding the *eventually* and *until* operators, and using negation only as in the negation restricted fragment. This fragment can be used for checking safety and bounded liveness properties.

All formulae in this fragment are closed, i.e. the infinite refusal traces are the closure of the finite ones. It is therefore convenient to restrict attention to the finite refusal traces by defining:

$$\mathcal{R}[\![\phi]\!] \mathrel{\widehat{=}} \mathcal{F}[\![\phi]\!] \cap RT,$$

and noting that

$$P \models \phi \ \textit{iff}\ \mathcal{R}[\![P]\!] \subseteq \mathcal{R}[\![\phi]\!].$$

4. Constructing the CSP specification

In this section we describe the construction of the CSP specification corresponding to a formula of the bounded, positive fragment, $BPTL$. The algorithm has been implemented in Haskell.[4]

We may assume that all occurrences of the negation operator are of the form $\neg a$; all other occurrences can be removed using equations such as $\textsf{true} \equiv \neg\textsf{false}$ and $\textsf{live} \equiv \neg\textsf{deadlocked}$.

4.1. Overview

The construction of the CSP specification corresponding to a formula ϕ proceeds in two stages. In the first stage, we build an automaton corresponding to ϕ. This automaton is constructed in much the same way as a Büchi automaton [VW86, Wol01]. States of the automaton are labelled with sets of formulae that must hold in that state.

In the second stage, the automaton is translated into a corresponding CSP process. States of the automaton are translated into states of the specification, with transitions corresponding to transitions of the automaton. In each state, appropriate events are available to ensure that the corresponding atomic formulae are satisfied.

For example, consider the formula

$$\phi \mathrel{\widehat{=}} (\bigcirc\,\textsf{available}\,a)\ \mathcal{R}\ \textsf{available}\,b.$$

The automaton for ϕ is in Fig. 1. The initial states are 0 and 1: in each of these ϕ holds. Note that

$$\phi = (\textsf{available}\,b \wedge \bigcirc\,\phi) \vee (\textsf{available}\,b \wedge \bigcirc\,\textsf{available}\,a),$$

because of the equation[5]

$$\psi\ \mathcal{R}\ \chi \equiv (\chi \wedge \bigcirc(\psi\ \mathcal{R}\ \chi)) \vee (\chi \wedge \psi). \tag{2}$$

The disjunction is split between the states: in state 0, $\textsf{available}\,b \wedge \bigcirc\,\phi$ holds, whereas in state 1, $\textsf{available}\,b \wedge \bigcirc\,\textsf{available}\,a$ holds. The former state must be succeeded by a state where ϕ still holds, i.e. 0 or 1. The latter state must be succeeded by a state where $\textsf{available}\,a$ holds, namely state 2. State 2 contains no *next* formula, so is succeeded by a state with no constraints, namely state 3.

[4] Implementation available from http://web.comlab.ox.ac.uk/oucl/work/gavin.lowe/Papers/temporalLogic.tar.

[5] Note that χ holds at the point of release, where ψ first becomes true, corresponding to the strict inequality in the semantic equation for $\mathcal{R}$.

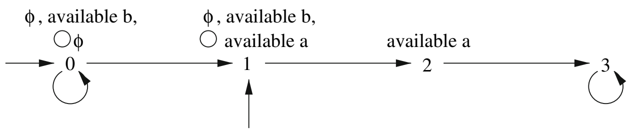

Fig. 1. Automaton for $\phi \mathrel{\widehat{=}} (\bigcirc \textsf{available}\, a)\ \mathcal{R}\ \textsf{available}\, b$

The corresponding specification process is

$$\begin{aligned}
\text{let } & Spec_0 = (STOP \sqcap \$x : \Sigma \to (Spec_0 \sqcap Spec_1)) \mathbin{\Box} b \to (Spec_0 \sqcap Spec_1)\\
& Spec_1 = (STOP \sqcap \$x : \Sigma \to Spec_2) \mathbin{\Box} b \to Spec_2\\
& Spec_2 = (STOP \sqcap \$x : \Sigma \to Spec_3) \mathbin{\Box} a \to Spec_3\\
& Spec_3 = (STOP \sqcap \$x : \Sigma \to Spec_3)\\
\text{within } & Spec_0 \sqcap Spec_1.
\end{aligned}$$

The subsidiary processes $Spec_0, \ldots, Spec_3$ correspond to the similarly-numbered states. The definitions ensure that $Spec_0$ and $Spec_1$ satisfy $\textsf{available}\, b$, and that $Spec_2$ satisfies $\textsf{available}\, a$; in each case, arbitrary other events may be performed. The succeeding states correspond to Fig. 1.

The rest of this section is organised as follows: in Sect. 4.2 we describe the construction of the automaton; in Sect. 4.3 we describe the desired relationship between the automaton and the corresponding CSP specification; and in Sect. 4.4 we describe how to build that CSP specification.

4.2. Building the automaton

In this section we describe in more detail how the automaton is built. The construction follows closely that of [VW86, Wol01].

Recall that a state of the automaton is labelled with a set Φ of formulae (of type $\mathbb{P}\, BPTL$). More precisely, it will be a *saturated set*, by which we mean a set where each of the formulae is either atomic (we treat $\neg a$ as being atomic) or of the form $\bigcirc \phi$.

The following function converts a set of formulae into a set of saturated sets, by splitting all disjuncts, taking Eq. (2) into account:

$$\begin{aligned}
\text{saturate} &: \mathbb{P}\, BPTL \to \mathbb{P}(\mathbb{P}\, BPTL)\\
\text{saturate}(\{\}) &= \{\{\}\},\\
\text{saturate}(\{\phi \wedge \psi\} \cup \Phi) &= \text{saturate}(\{\phi, \psi\} \cup \Phi),\\
\text{saturate}(\{\phi \vee \psi\} \cup \Phi) &= \text{saturate}(\{\phi\} \cup \Phi) \cup \text{saturate}(\{\psi\} \cup \Phi),\\
\text{saturate}(\{\phi\ \mathcal{R}\ \psi\} \cup \Phi) &= \text{saturate}(\{\phi, \psi\} \cup \Phi) \cup\ \text{saturate}(\{\psi, \bigcirc(\phi\ \mathcal{R}\ \psi)\} \cup \Phi),\\
\text{saturate}(\{\Box\, \phi\} \cup \Phi) &= \text{saturate}(\{\phi, \bigcirc \Box\, \phi\} \cup \Phi)\\
\text{saturate}(\{\phi\} \cup \Phi) &= \{\{\phi\} \cup \Psi \mid \Psi \in \text{saturate}(\Phi)\}, \qquad \text{otherwise.}
\end{aligned}$$

Note that the order of application of the above rules does not matter.

For example, taking the example ϕ from the previous subsection, we have

$$\text{saturate}(\phi) = \{\{\bigcirc(\textsf{available}\, a), \textsf{available}\, b\}, \{\textsf{available}\, b, \bigcirc \phi\}\}.$$

A set of formulae Φ represents the conjunction of those formulae, $\bigwedge \Phi$. The following lemma relates this to the result of saturating Φ:

Lemma 3 $\bigwedge \Phi \equiv \bigvee\{\bigwedge \Psi \mid \Psi \in \text{saturate}(\Phi)\}$.

Given an initial formula ϕ, the algorithm to construct the automaton starts with the set of states saturate($\{\phi\}$). For a saturated set Φ, we write nexts(Φ) for all the constraints on the next state:

$$\text{nexts}(\Phi) \mathrel{\widehat{=}} \{\psi \mid \bigcirc \psi \in \Phi\}.$$

The automaton includes a transition from state Φ to each state Ψ in saturate(nexts(Φ)). The algorithm continues by building transitions from each of those states Ψ. The reader might like to verify that the automaton for the ϕ from the previous subsection is indeed as in Fig. 1.

The following lemma relates the formulae of a state to its atomic formulae and the formulae of successive states.

Lemma 4 If Φ is saturated, then $\bigwedge \Phi \equiv \bigwedge \text{atoms}(\Phi) \wedge \bigcirc \bigvee \{\bigwedge \Psi \mid \Phi \rightarrow \Psi\}$, where atoms($\Phi$) represents the atomic formulae in Φ, and $\rightarrow$ represents the transition relation in the automaton.

Proof. We have:

$$\begin{array}{l} \quad \bigwedge \Phi \\ \equiv \langle\ \Phi \text{ is saturated} \rangle \\ \quad \bigwedge \text{atoms}(\Phi) \wedge \bigwedge \{\bigcirc \phi \mid \bigcirc \phi \in \Phi\} \\ \equiv \langle\ \bigcirc \text{ distributes over conjunction} \rangle \\ \quad \bigwedge \text{atoms}(\Phi) \wedge \bigcirc (\bigwedge \text{nexts}(\Phi)) \\ \equiv \langle\ \text{Lemma } 3 \rangle \\ \quad \bigwedge \text{atoms}(\Phi) \wedge \bigcirc (\bigvee \{\bigwedge \Psi \mid \Psi \in \text{saturate}(\text{nexts}(\Phi))\}) \\ \equiv \langle\ \text{from the construction of the automaton} \rangle \\ \quad \bigwedge \text{atoms}(\Phi) \wedge \bigcirc (\bigvee \{\bigwedge \Psi \mid \Phi \rightarrow \Psi\}). \end{array}$$

□

We can make two optimisations to the algorithm.

1. After saturating, we remove states Φ that are logically equivalent to false; this will be the case if, for example:
   (a) Φ contains false;
   (b) Φ contains a and b, for $a \neq b$;
   (c) Φ contains a and $\neg a$;
   (d) Φ contains deadlocked and either live, a or available a.
2. If state Φ includes transitions to Ψ and Ψ' such that Ψ' logically implies Ψ, i.e. $\mathcal{R}[\![\bigwedge \Psi]\!] \supseteq \mathcal{R}[\![\bigwedge \Psi']\!]$, then we can remove the transition to Ψ', because all the behaviours of Ψ' are included in Ψ. This will be the case when every formula $\psi \in \Psi$ is implied by some formula $\psi' \in \Psi'$, for example if:
   (a) $\psi = \text{available}\, a$ and $\psi' = a$;
   (b) $\psi = \text{live}$ and $\psi' = a$;
   (c) $\psi = \neg a$ and $\psi' = \text{deadlocked}$.

4.3. Requirements for the specification process

In this section we consider how the CSP specification process should relate to the corresponding temporal logic specification. The obvious thing to do would be to ensure that $\mathcal{R}[\![Spec(\phi)]\!] = \mathcal{R}[\![\phi]\!]$. However, that turns out not to be possible.

Consider the temporal logic formula $\phi \mathrel{\widehat{=}} a \wedge \text{available}\, b$. Then

$$\mathcal{R}[\![\phi]\!] = \{\langle\rangle\} \cup \{\langle X, a\rangle^\frown tr \mid a, b \notin X \wedge tr \in RT\}.$$

No specification process has precisely those refusal traces: by Condition R4, if a process cannot refuse b initially, and stabilises, then it must be able to perform a b. The specification process we will use for ϕ will be $a \rightarrow Chaos \,\Box\, b \rightarrow \text{div}$. This slightly overestimates the refusal traces of ϕ: it has the additional refusal traces $\{\langle X, b\rangle \mid a, b \notin X\}$.

Note that $a \wedge \text{available}\, b$ is infeasible: any process that satisfies it must be able to perform b initially (because b must be available), but that contradicts the first conjunct. Likewise, the CSP specification $a \rightarrow Chaos \,\Box\, b \rightarrow \text{div}$ is infeasible (for divergence-free processes): any process P that refines it must be able to perform b initially, say with initial behaviour $\langle X, b\rangle$; but then, by Eq. (1), P must have some extension of $\langle X, b\rangle$, which is not allowed by the specification process. However, ϕ is not equivalent to false. For example, $\phi \vee b$ is satisfied by $a \rightarrow STOP \rhd b \rightarrow STOP$ and has specification process $(a \rightarrow Chaos \,\Box\, b \rightarrow \text{div}) \sqcap b \rightarrow Chaos$.

More generally, we will have

$$\mathcal{R}[\![\phi]\!] \subseteq \mathcal{R}[\![\mathit{Spec}(\phi)]\!] \subseteq \mathcal{R}[\![\phi]\!] \cup \{tr^\frown\langle X, a\rangle \mid tr \in \mathcal{R}[\![\phi]\!] \wedge \langle X, a\rangle \in RT_1\}. \tag{3}$$

In other words, $\mathcal{R}[\![\mathit{Spec}(\phi)]\!]$ may overestimate $\mathcal{R}[\![\phi]\!]$, but only by single non-deadlocking steps. The following lemma shows that this condition gives us what we want.

Lemma 5 Suppose $\mathit{Spec}(\phi)$ satisfies Eq. (3), and suppose P is divergence-free. Then $P \models \phi \Leftrightarrow \mathit{Spec}(\phi) \sqsubseteq P$.

Proof. For the left-to-right direction, if $P \models \phi$ then $\mathcal{R}[\![P]\!] \subseteq \mathcal{R}[\![\phi]\!]$, so $\mathcal{R}[\![P]\!] \subseteq \mathcal{R}[\![\mathit{Spec}(\phi)]\!]$ by Eq. (3), so $\mathit{Spec}(\phi) \sqsubseteq P$.

Conversely, suppose $\mathit{Spec}(\phi) \sqsubseteq P$, so $\mathcal{R}[\![P]\!] \subseteq \mathcal{R}[\![\mathit{Spec}(\phi)]\!]$. Consider $tr \in \mathcal{R}[\![P]\!]$, so $tr \in \mathcal{R}[\![\mathit{Spec}(\phi)]\!]$. Suppose, for a contradiction, that $tr \notin \mathcal{R}[\![\phi]\!]$. Then by Eq. (3), tr is of the form $tr'^\frown\langle X, a\rangle$. But P is divergence free, so by Eq. (1), some extension tr'' of tr is in $\mathcal{R}[\![P]\!]$: tr'' is of the form (i) $tr^\frown\langle\Sigma\rangle$ or (ii) $tr^\frown\langle\bullet, c\rangle$. Further, $\mathit{Spec}(\phi) \sqsubseteq P$, so $tr'' \in \mathcal{R}[\![\mathit{Spec}(\phi)]\!]$. If $tr'' \in \mathcal{R}[\![\phi]\!]$, then $tr \in \mathcal{R}[\![\phi]\!]$, because $\mathcal{R}[\![\phi]\!]$ is prefix-closed by Lemma 2, giving a contradiction. Alternatively, if $tr'' \notin \mathcal{R}[\![\phi]\!]$, then tr'' must come from the set comprehension in Eq. (3), and so must be of form (ii), and $tr \in \mathcal{R}[\![\phi]\!]$, giving a contradiction. □

We will build a specification process $\mathit{Spec}(\bigwedge \Phi)$ for each state Φ of the automaton; we will tend to abuse notation and write $\mathit{Spec}(\Phi)$. Given initial specification ϕ, the initial states of the automaton are *saturate*($\{\phi\}$), so the specification process will be

$$\mathit{Spec}(\phi) \mathrel{\widehat{=}} \textstyle\bigsqcap_{\Phi \in \mathrm{saturate}(\{\phi\})} \mathit{Spec}(\Phi).$$

Each $\mathit{Spec}(\Phi)$ will be such that the initial behaviour will include the initial behaviour of Φ, and the subsequent behaviour will be that of those $\mathit{Spec}(\Psi)$ such that $\Phi \rightarrow \Psi$, i.e.[6] $\bigsqcap_{\Psi \mid \Phi \rightarrow \Psi} \mathit{Spec}(\Psi)$.

Given a set S of refusal traces, we write inits(S) for the initial behaviours of S:

$$\mathrm{inits}(S) \mathrel{\widehat{=}} (\text{if } \langle\Sigma\rangle \in S \text{ then } \{\langle\Sigma\rangle\} \text{ else } \{\}) \cup \{\langle X, a\rangle \mid \langle X, a\rangle^\frown tr \in S\}.$$

We write inits(Spec) for inits($\mathcal{R}[\![\mathit{Spec}]\!]$), inits($\Phi$) for inits($\mathcal{R}[\![\bigwedge \Phi]\!]$), etc. If $\langle X, a\rangle \in \mathrm{inits}(P)$, then we write P after $\langle X, a\rangle$ for the process that behaves like P after it has performed $\langle X, a\rangle$, i.e. the process with behaviours $\{tr \mid \langle X, a\rangle^\frown tr \in \mathcal{R}[\![P]\!]\}$.

The following lemma gives sufficient conditions on $\mathit{Spec}(\Phi)$ for it to be used for testing for the satisfaction of $\bigwedge \Phi$.

Lemma 6 Suppose, for every Φ, the initials of $\mathit{Spec}(\Phi)$ include the initials of Φ and possibly some non-deadlocked behaviours:

$$\mathrm{inits}(\Phi) \subseteq \mathrm{inits}(\mathit{Spec}(\Phi)) \subseteq \mathrm{inits}(\Phi) \cup RT_1, \tag{4}$$

and the behaviour of $\mathit{Spec}(\Phi)$ after an initial of Φ corresponds to the automaton:

$$\forall \langle X, a\rangle \in \mathrm{inits}(\Phi) \bullet \mathit{Spec}(\Phi) \;\mathsf{after}\; \langle X, a\rangle = \textstyle\bigsqcap_{\Psi \mid \Phi \rightarrow \Psi} \mathit{Spec}(\Psi), \tag{5}$$

and the behaviour of $\mathit{Spec}(\Phi)$ after any other initial is div:

$$\forall \langle X, a\rangle \in \mathrm{inits}(\mathit{Spec}(\Phi)) - \mathrm{inits}(\Phi) \bullet \mathit{Spec}(\Phi) \;\mathsf{after}\; \langle X, a\rangle = \mathsf{div}. \tag{6}$$

Then Eq. (3) is satisfied, with ϕ replaced by $\bigwedge \Phi$:

$$\mathcal{R}[\![\textstyle\bigwedge \Phi]\!] \subseteq \mathcal{R}[\![\mathit{Spec}(\Phi)]\!] \subseteq \mathcal{R}[\![\bigwedge \Phi]\!] \cup \{tr^\frown\langle X, a\rangle \mid tr \in \mathcal{R}[\![\bigwedge \Phi]\!] \wedge \langle X, a\rangle \in RT_1\}. \tag{7}$$

So by Lemma 5, $\mathit{Spec}(\Phi)$ captures the requirements of Φ for refinement purposes:

$$P \models \textstyle\bigwedge \Phi \Leftrightarrow \mathit{Spec}(\Phi) \sqsubseteq P.$$

[6] It is possible that this expression represents a nondeterministic choice over the empty set, if all such transitions were removed by optimisation 1 of Sect. 4.2. Normally, the nondeterministic choice over an empty set is undefined. For the purposes of this paper, we define it to be div; this definition makes sense, because div is a unit of nondeterministic choice. FDR does not follow this convention, so the implementation of the translation treats this as a special case.

Proof. For the first part of Eq. (7), suppose $tr \in \mathcal{R}[\![\bigwedge \Phi]\!]$; we show $tr \in \mathcal{R}[\![Spec(\Phi)]\!]$ by induction on tr. The case $tr = \langle\rangle$ follows from condition R1. The case $tr = \langle\Sigma\rangle$ follows from Eq. (4). For $tr = \langle X, a\rangle^\frown tr'$, we have $\langle X, a\rangle \in \text{inits}(Spec(\Phi))$ by Eq. (4). Further, by Lemma 4, $tr' \in \mathcal{R}[\![\bigwedge \Psi]\!]$ for some Ψ such that $\Phi \to \Psi$. Hence, by the inductive hypothesis, $tr' \in \mathcal{R}[\![Spec(\Psi)]\!]$, and hence by (5), $tr \in \mathcal{R}[\![Spec(\Phi)]\!]$.

For the second part of Eq. (7), suppose $tr \in \mathcal{R}[\![Spec(\Phi)]\!]$; we proceed by induction on tr. The cases $tr = \langle\rangle$ and $tr = \langle\Sigma\rangle$ follow from Lemma 1 and Eq. (4), respectively. For $tr = \langle X, a\rangle^\frown tr'$, we consider two cases:

- Case $\langle X, a\rangle \in \text{inits}(\Phi)$. Then by Eq. (5), $tr' \in \mathcal{R}[\![Spec(\bigwedge \Psi)]\!]$ for some Ψ such that $\Phi \to \Psi$. Then by the inductive hypothesis, $tr' \in \mathcal{R}[\![\bigwedge \Psi]\!] \cup \{tr''^\frown\langle Y, b\rangle \mid tr'' \in \mathcal{R}[\![\bigwedge \Psi]\!]\}$. If $tr' \in \mathcal{R}[\![\bigwedge \Psi]\!]$ then $tr \in \mathcal{R}[\![\bigwedge \Phi]\!]$ by Lemma 4, as required. Otherwise, tr' is of the form $tr''^\frown\langle Y, b\rangle$ where $tr'' \in \mathcal{R}[\![\bigwedge \Psi]\!]$. Then $\langle X, a\rangle^\frown tr'' \in \mathcal{R}[\![\bigwedge \Phi]\!]$ by Lemma 4, and so $tr = \langle X, a\rangle^\frown tr''^\frown\langle Y, b\rangle \in \{tr'''^\frown\langle Y, b\rangle \mid tr''' \in \mathcal{R}[\![\bigwedge \Phi]\!]\}$, as required.
- Case $\langle X, a\rangle \notin \text{inits}(\Phi)$. Then by Eq. (6), $tr' = \langle\rangle$, so $tr = \langle X, a\rangle$ and so $tr \in \{tr''^\frown\langle X, a\rangle \mid tr'' \in \mathcal{R}[\![\bigwedge \Phi]\!]\}$, since $\langle\rangle \in \mathcal{R}[\![\bigwedge \Phi]\!]$ by Lemma 1 (R1).

This completes the proof. □

4.4. Building the specification process

In this section we explain how to build the $Spec(\Phi)$ processes, for saturated sets Φ. Following Lemma 6, we need to ensure that:

- $Spec(\Phi)$ has initial behaviour including the initials of Φ and possibly some non-deadlocked behaviours [Eq. (4)]; in fact, we ensure

  $$\text{inits}(Spec(\Phi)) = \text{inits}(\Phi) \cup \{\langle X, a\rangle \mid \exists b \bullet \langle X, b\rangle \in \text{inits}(\Phi) \wedge \langle X \cup \{a\}, b\rangle \notin \text{inits}(\Phi)\},$$

  i.e. the smallest superset of inits(Φ) that allows the performance of those events required by condition R4.
- After initial behaviours of Φ, $Spec(\Phi)$ evolves corresponding to the automaton [Eq. (5)], i.e. to the process

  $$Nexts \mathrel{\hat=} \textstyle\sqcap_{\Psi \mid \Phi \to \Psi} Spec(\Psi).$$
- After other behaviours, $Spec(\Phi)$ evolves to div [Eq. (6)].

Suppose Φ contains formulae a for each $a \in A$, $\mathsf{available}\, b$ for each $b \in B$, and $\neg c$ for each $c \in C$ (where $A, B, C \subseteq \Sigma$). Because of optimisation (1.b) of Sect. 4.2, we will have $\#A \leq 1$, and because of optimisation (1.c), $A \cap C = \{\}$. Further, $\mathsf{false} \notin \Phi$ because of optimisation (1.a). We perform a case analysis.

Case $\#A = 1$, say $A = \{a\}$. Because of optimisation (1.d), $\mathsf{deadlocked} \notin A$. We can calculate as below; note that $\text{inits}(a) \subseteq \text{inits}(\mathsf{live})$, so it makes no difference whether or not live is in Φ:

$$\begin{aligned}\text{inits}(\Phi) &= \text{inits}(a) \cap \textstyle\bigcap_{b \in B} \text{inits}(\mathsf{available}\, b) \cap \bigcap_{c \in C} \text{inits}(\neg c) \\ &= \{\langle X, a\rangle \mid a \notin X\} \cap \textstyle\bigcap_{b \in B}\{\langle X, x\rangle \mid b \notin X \wedge x \notin X\} \cap \\ &\quad \textstyle\bigcap_{c \in C}(\{\langle\Sigma\rangle\} \cup \{\langle X, x\rangle \mid x \neq c \wedge x \notin X\}) \\ &= \{\langle X, a\rangle \mid X \cap (\{a\} \cup B) = \{\}\}.\end{aligned}$$

The corresponding CSP specification process will be

$$Spec(\Phi) \mathrel{\hat=} a \to Nexts \;\Box\; ?b : B \to \mathsf{div}.$$

Then

$$\text{inits}(Spec(\Phi)) = \{\langle X, a\rangle \mid X \cap (\{a\} \cup B) = \{\}\} \cup \{\langle X, b\rangle \mid X \cap (\{a\} \cup B) = \{\} \wedge b \in B\}.$$

It is straightforward to check that Eqs. (4)–(6) hold.

Case $A = \{\}$ and $\mathsf{deadlocked} \in \Phi$. Because of optimisation (1.d), Φ contains no formula of the form $\mathsf{available}\, a$ or live. Hence $\text{inits}(\Phi) = \{\langle\Sigma\rangle\}$. The corresponding specification is $STOP$, and $\text{inits}(STOP) = \{\langle\Sigma\rangle\}$, so Eqs. (4)–(6) hold.

Case $A = \{\}$, $\mathsf{deadlocked} \notin \Phi$ and $\mathsf{live} \notin \Phi$. Then

$$\text{inits}(\Phi) = (\text{if } B = \{\} \text{ then } \{\langle\Sigma\rangle\} \text{ else } \{\}) \cup \{\langle X, x\rangle \mid X \cap B = \{\} \wedge x \in \Sigma - (C \cup X)\}.$$

The corresponding specification is

$$Spec(\Phi) \mathrel{\hat{=}} (STOP \sqcap \$x : \Sigma - C \rightarrow Nexts) \mathrel{\Box} ?b : B \rightarrow \text{if } b \in C \text{ then } \mathsf{div} \text{ else } Nexts.$$

Then

$$\begin{aligned}\text{inits}(Spec(\Phi)) = {} & (\text{if } B = \{\} \text{ then } \{\langle\Sigma\rangle\} \text{ else } \{\}) \cup \\ & \{\langle X, x\rangle \mid X \cap B = \{\} \wedge x \in \Sigma - (C \cup X)\} \cup \\ & \{\langle X, b\rangle \mid X \cap B = \{\} \wedge b \in B - X\}.\end{aligned}$$

It is straightforward to check that Eqs. (4)–(6) hold.

Case $A = \{\}$**,** $\mathsf{deadlocked} \notin \Phi$ **and** $\mathsf{live} \in \Phi$**.** If $B \neq \{\}$, then this case reduces to the previous one. If $B = \{\}$ then

$$\text{inits}(\Phi) = \{\langle X, x\rangle \mid x \in \Sigma - (C \cup X)\}.$$

The corresponding specification is

$$Spec(\Phi) \mathrel{\hat{=}} \$x : \Sigma - C \rightarrow Nexts.$$

Then $\text{inits}(Spec(\Phi)) = \{\langle X, x\rangle \mid x \in \Sigma - (C \cup X)\}$, and Eqs. (4)–(6) hold.

This completes the construction of the CSP specification process.

The reader might like to verify that for the example ϕ from Sect. 4.1, the specification process is indeed as given there.

As a slightly larger example, consider the formula

$$a \wedge \mathsf{available}\, b \wedge \neg c \wedge O(\mathsf{available}\, b \wedge \mathsf{available}\, c \wedge \neg c \wedge O(\mathsf{live} \wedge \neg c \wedge O(\mathsf{deadlocked} \wedge \neg c))).$$

This produces the following specification process

$$\begin{aligned}&\text{let } Spec_0 = a \rightarrow Spec_1 \mathrel{\Box} b \rightarrow \mathsf{div}\\ &\quad Spec_1 = (STOP \sqcap \$x : \Sigma - \{c\} \rightarrow Spec_2) \mathrel{\Box} b \rightarrow Spec_2 \mathrel{\Box} c \rightarrow \mathsf{div}\\ &\quad Spec_2 = \$x : \Sigma - \{c\} \rightarrow Spec_3\\ &\quad Spec_3 = STOP\\ &\text{within } Spec_0\end{aligned}$$

The states of the specification process illustrate, respectively, the first, third, fourth and second cases, above.

In general, the number of states of the automaton, and hence the specification process, is exponential in the size of the formula [Wol01]. However, in practice the automaton is normally much smaller. For all the examples I have tried, the implementation has produced the specification process effectively instantaneously.

5. A characterisation theorem

One question we have not yet addressed is that of the expressivity of our logic with respect to the (finite) refusal traces model. We can ask whether the refusal traces model is characterised by the logic: given two processes P and Q that are distinguished by the model, can we can find a formula ϕ that distinguishes them, i.e. such that $P \models \phi$ and $Q \not\models \phi$, or vice versa? In this section we show that this is not quite true, but that a small addition to the logic makes it true.

Consider the two processes:

$$P \mathrel{\hat{=}} (a \rightarrow b \rightarrow STOP \sqcap a \rightarrow c \rightarrow STOP) \mathrel{\Box} ?x : \Sigma - \{a\} \rightarrow STOP,$$

$$Q \mathrel{\hat{=}} (a \rightarrow b \rightarrow STOP \rhd a \rightarrow c \rightarrow STOP) \mathrel{\Box} ?x : \Sigma - \{a\} \rightarrow STOP.$$

Semantically, the only difference is that P has refusal traces of the form $\langle\{\}, a, X, b\rangle$ for $b \notin X$, which Q does not; i.e. P can stabilise before performing the a that leads to a b.

It turns out that no formula can distinguish these processes. This follows from the following lemma, which shows that formulae of the logic cannot detect instability.

Lemma 7 For all formulae $\phi \in TL$, and for all refusal traces $tr \in PRT$ and $tr' \in RT$,

$$tr\frown\langle\{\}, a\rangle\frown tr' \in \mathcal{R}[\![\phi]\!] \Leftrightarrow tr\frown\langle\bullet, a\rangle\frown tr' \in \mathcal{R}[\![\phi]\!].$$

Any formula ϕ that distinguishes P and Q would have to allow all refusal traces common to both, including $\langle \bullet, a, X, b \rangle$; the lemma then says that ϕ would also have to allow $\langle \{\}, a, X, b \rangle$, and so would not distinguish P and Q.

Let us consider the effect of adding an atomic predicate unstable that says that the process does not stabilise initially:

$$\phi ::= \ldots \mid \mathsf{unstable}.$$

The semantics of unstable is given by:

$$\mathcal{R}[\![\mathsf{unstable}]\!] \mathrel{\widehat{=}} \{\langle\rangle\} \cup \{\langle \bullet, b \rangle ^\frown tr \mid b \in \Sigma\}.$$

Returning to the above example, if we define

$$\phi \mathrel{\widehat{=}} \mathsf{unstable} \vee \neg a \vee \bigcirc \neg b,$$

then $Q \models \phi$, but $P \not\models \phi$.

The following theorem shows that this addition is enough.

Theorem 8 The refusal traces model is characterised by *BPTL* with the addition of the unstable formula.

Proof. Suppose P and Q are distinguished by the (finite) refusal traces model. Then, without loss of generality, suppose there is some refusal trace $tr \in \mathcal{R}[\![P]\!] - \mathcal{R}[\![Q]\!]$. We perform a case analysis:

- Case tr is a partial trace, say $tr = \langle X_0, a_0, \ldots, X_n, a_n \rangle$. We construct a formula ϕ such that $Q \models \phi$ and $P \not\models \phi$. We need first to define a macro that says that a particular refusal token and all larger ones are not possible:

  $$\begin{aligned} \texttt{NOT-REF}\{x_1, \ldots, x_m\} &\mathrel{\widehat{=}} \mathsf{unstable} \vee \mathsf{available}\, x_1 \vee \ldots \vee \mathsf{available}\, x_m, \\ \texttt{NOT-REF}\,\bullet &\mathrel{\widehat{=}} \mathsf{false}. \end{aligned}$$

  (The unstable disjunct is unnecessary for $m \geq 1$, since it is included in the other terms). Note that $\mathcal{R}[\![\,\texttt{NOT-REF}\,X]\!]$ contains all refusal traces except $\langle \Sigma \rangle$ and those of the form $\langle Y, a \rangle ^\frown tr$ with $Y \supseteq X$.
  We define ϕ by:

  $$\phi \mathrel{\widehat{=}} \texttt{NOT-REF}\,X_0 \vee \neg a_0 \vee \bigcirc(\texttt{NOT-REF}\,X_1 \vee \neg a_1) \vee \ldots \vee \bigcirc^n(\texttt{NOT-REF}\,X_n \vee \neg a_n),$$

  (where $\bigcirc^n$ represents n applications of the $\bigcirc$ operator). Then all traces *not* allowed by ϕ are of the form $\langle Y_0, a_0, \ldots, Y_n, a_n \rangle ^\frown tr'$ where $Y_i \supseteq X_i$ for $i = 0, \ldots, n$. Now, Q does not have refusal trace tr, so by healthiness conditions 2 and 3 has no trace of the above form, and so $Q \models \phi$. And P has trace tr so $P \not\models \phi$.
- Case tr is a deadlocked trace, say $tr = \langle X_0, a_0, \ldots, X_n, a_n, \Sigma \rangle$. We define ϕ by:

  $$\phi \mathrel{\widehat{=}} \texttt{NOT-REF}\,X_0 \vee \neg a_0 \vee \bigcirc(\texttt{NOT-REF}\,X_1 \vee \neg a_1) \vee \ldots \vee \bigcirc^n(\texttt{NOT-REF}\,X_n \vee \neg a_n) \vee \bigcirc^{n+1}\,\mathsf{live}.$$

  Similarly to the previous case, we have $P \not\models \phi$ and $Q \models \phi$. □

In fact, the above proof shows that the refusal traces model is characterised by a logic with grammar:

$$\phi ::= \mathsf{available}\, a \mid \neg a \mid \mathsf{unstable} \mid \phi \vee \phi \mid \bigcirc \phi.$$

(The live term in the second case of the proof can be replaced by $\bigvee_{a \in \Sigma} \mathsf{available}\, a$).

We do not consider the unstable formula a practically useful one, which is why we have not included it earlier. However, adding it to the construction of the specification process is straightforward: we can adapt the process in Sect. 4.4 so as to force appropriate states to be unstable by adding a sliding choice to div.

It is worth noting that, while it is possible to test for *in*stability, it is not possible to test for stability, since stability is not preserved by refinement.

6. Conclusions

In this paper, we have considered the relationship between temporal logic-based and refinement-based model checking, within the context of communicating systems, as typically modelled using a process algebra such as CSP. We have considered what atomic formulae are suitable in such a setting. We have shown that the standard stable failures model of CSP is not suitable for capturing such formulae, and instead the refusal traces model is appropriate. We have captured the semantics of the logic within the (infinite) refusal traces model. We have also shown that it is not possible, in general, to check for satisfaction of formulae using the *eventually*, *until* or negation operators via simple model checking.

We have shown that the remaining fragment (the bounded, positive fragment) of the logic can be checked in this way: we have described how to produce a suitable CSP specification, by first constructing a Büchi-like automaton, and then translating that automaton into a CSP specification, where the initial actions and subsequent behaviour respect the atomic formulae and transitions of the automaton, respectively.

Finally, we have shown that, with a small addition, the refusal traces model is characterised by the bounded, positive fragment.

The overhead of creating specifications in this way—translating from the temporal logic, as opposed to writing the refinement-oriented specification direct—is negligible compared to the time spent on other aspects of the model checking. Both the translation described in the previous section, and the normalisation of the specification (see [Ros97, Appendix C]) are effectively instantaneous.

6.1. Related work

In [LMC01], Leuschel, Massart and Currie study the same problem as us: they translate satisfaction of LTL formulae into CSP refinement checks. Their approach has two major differences from ours.

Firstly, they do not include formulae of the form $\mathsf{available}\,a$. A consequence of this is that they are able to talk only about the performance or non-performance of events, rather than the availability of events. They are therefore unable to distinguish between nondeterministic and environmental choices, so can deal only with properties of traces.

Further, the form of their refinement check is such that

$$P \models \phi \;\text{ iff }\; (P \parallel \mathit{Tester}(\phi))\backslash(\Sigma - \{\mathit{success}\}) \not\sqsubseteq_T \mathit{SUC}, \tag{8}$$

where P is the process in question, $\mathit{Tester}(\phi)$ is a tester process, depending on the specification ϕ, and $\mathit{SUC} = \mathit{success} \to \mathit{SUC}$ is a fixed process. Note that the process P appears on the *left* of the refinement check. This severely limits the applicability of the technique: FDR works by normalising the left hand side of a refinement check, a technique that is, in the worst case, exponential in that process's state space. By contrast, refinement checking is linear in the state space of the right hand side. Hence FDR is normally used on refinement checks with a small left hand side, but potentially a much larger right hand side (up to about 10^{10} states).

Roscoe [Ros05] considers the expressive power of CSP refinement checks. More precisely, he considers which predicates over the failures-divergences model may be captured by refinement checks of the form $F(P) \sqsubseteq G(P)$ where F, G are uniformly continuous CSP contexts. He shows that all predicates that are closed (under the normal topology over the failures divergences model [Ros97]) can be captured in this way, and vice versa. He further shows that every satisfiable, refinement-closed and distributive predicate may be captured as a *simple* refinement check, and vice versa. Note, however, that he does not restrict himself to finitary contexts (i.e. $F(P)$ and $G(P)$ may be infinite state when P is finite): hence that paper is complementary to the current paper, asking what is theoretically possible rather than giving practical techniques. It would be interesting to investigate whether Roscoe's results carry over to the refusal traces model used in the current paper.

Various other papers have considered the relationship between process algebras and modal logics. Hennessy and Milner [HM85] give a characterisation of bisimulation equivalence in terms of a modal logic, subsequently referred to as *Hennessy-Milner Logic*. Aceto and Ingólfsdóttir [AI99] show that the properties checkable by testing are precisely the safety properties of Hennessy-Milner Logic extended with recursion. Boudon and Larsen [BL92] investigate (bisimulation-based) refinement of labelled transition systems with two kinds of transitions, those that must be available, and those that may be available; they show that the properties captured by such refinements are precisely the prime, satisfiable formulae of Hennessy-Milner Logic, i.e. those formulae ϕ such that if ϕ implies $\phi_0 \vee \phi_1$ then ϕ implies ϕ_0 or ϕ implies ϕ_1.

6.2. Future work

We have shown that we cannot check for *eventually* or *until* properties using simple model checking; the obvious question, therefore, is whether some more general refinement check will suffice. Given the above discussion about normal FDR operation, we want to restrict ourselves to checks where the process in question appears only on the right hand side of the refinement check. More precisely, given ϕ, we would like to construct a specification process $Spec_\phi$ and a CSP context F_ϕ such that

$$P \models \phi \text{ iff } Spec_\phi \sqsubseteq F_\phi(P).$$

(It is possible to transform Eq. (8) so that the left hand side is independent of P, using the technique of a watchdog specification [ZMGW02]; however, that again requires normalising the left hand side of (8), so doesn't gain anything; further, the right hand side of the new refinement check is not of the form $F_\phi(P)$ for a CSP context F_ϕ).

The argument of Sect. 3.5 can be extended to show that for certain specifications, such as $\Diamond a$, the context F_ϕ needs to use hiding. There are certainly tricks that one can play using hiding, and checking for the absence of divergence. For example:

$$P \models \Diamond a \text{ iff } a \rightarrow \mathsf{div} \sqsubseteq_{RD} P \backslash (\Sigma - \{a\}),$$
$$P \models \Box \Diamond a \text{ iff } Run(a) \sqsubseteq_{RD} P \backslash (\Sigma - \{a\}),$$

(where $Run(a) = a \rightarrow Run(a)$). These refinement checks are in the refusal traces/divergences model; the first refinement specifies that $P\backslash(\Sigma - \{a\})$ cannot diverge or deadlock before performing an a, i.e. that P performs an a after a finite number of other events; the second refinement specifies that $P\backslash(\Sigma - \{a\})$ cannot diverge or deadlock, but must perform infinitely many as, i.e. that P performs infinitely many as. However, these techniques seem difficult to generalise. In particular, Bill Roscoe has shown[7] that no such refinement check can check for the specification $\Diamond \Box a$.

NPATRL [SM96] is a temporal logic for specifying requirements for security protocols. I would like to extend the translation in this paper to cover NPATRL. I would then like to integrate this within Casper [Low98], the compiler that produces CSP models of security protocols, thereby providing a more flexible mechanism for specifying the requirements of protocols.

Acknowledgments

I would like to thank Joël Ouaknine, Bill Roscoe, James Worrell and the anonymous referees for useful comments and suggestions on this work.

References

[AI99] Aceto L, Ingólfsdóttir A (1999) Testing Hennessy–Milner logic with recursion. In: Foundations of software science and computation structure, pp 41–55
[BL92] Boudol G, Larsen KG (1992) Graphical versus logical specifications. Theor Comput Sci 106(1):3–20
[CES86] Clarke EM, Emerson EA, Sistla AP (1986) Automatic verification of finite-state concurrent systems using temporal logic specifications. ACM Trans Program Lang Syst 8(2):244–263
[HM85] Hennessy M, Milner R (1985) Algebraic laws for nondeterminism and concurrency. J ACM 32:137–161
[Hoa85] Hoare CAR (1985) Communicating sequential processes. Prentice Hall, Englewood Cliffs
[Hol97] Holzmann G (1997) The model checker SPIN. IEEE Trans Softw Eng 23(5):279–295
[Hol03] Holzmann G (2003) The SPIN model checker. Addison-Wesley, Reading
[Jac92] Jackson D (1992) Logic verification of reactive software systems. D. Phil thesis, Oxford University
[LMC01] Leuschel M, Massart T, Currie A (2001) How to make FDR spin: LTL model checking of CSP by refinement. In: Proceedings of formal methods Europe FME'2001, LNCS 2021. Springer, Heidelberg, pp 99–118
[Low98] Lowe G (1998) Casper: A compiler for the analysis of security protocols. J Comput Secur 6:53–84
[Muk93] Mukarram A (1993) A refusal testing model for CSP. D. Phil thesis, Oxford
[Phi87] Phillips I (1987) Refusal testing. Theor Comput Sci 50:241–284
[Pnu81] Pnueli A (1981) The temporal logic of concurrent programs. Theor Comput Sci 13:45–60
[Ros94] Roscoe AW (1994) Model-checking CSP. In: A classical mind, essays in honour of C.A.R. Hoare. Prentice-Hall, Englewood Cliffs
[Ros97] Roscoe AW (1997) The theory and practice of concurrency. Prentice Hall, Englewood Cliffs
[Ros05] Roscoe AW (2005) On the expressive power of CSP refinement. Form Aspects Comput 17(2):93–112
[SM96] Syverson P, Meadows C (1996) A formal language for cryptographic protocol requirements. Des Codes Cryptogr 7(1,2):27–59
[VW86] Vardi M, Wolper P (1986) An automata-theoretic approach to automatic program validation. In: Proceedings of LICS'86, pp 332–344

[7] personal communication.

[Wol83] Wolper P (1983) Temporal logic can be more expressive. Inf Control 56(1–2):72–99

[Wol01] Wolper P (2001) Constructing automata from temporal logic formulas: A tutorial. In: Lectures on formal methods in performance analysis (First EEF/Euro Summer School on Trends in Computer Science), LNCS 2090. Springer, Heidelberg, pp 261–277

[ZMGW02] Zakiuddin I, Moffat N, Goldsmith M, Whitworth T (2002) Property based compression strategies. In: Proceedings of the second workshop on automated verification of critical systems (AVoCS 2002)

Received 21 September 2006
Accepted in revised form 28 November 2007 by M. J. Butler
Published online 4 January 2008