Granularity and Concurrent Separation Logic

Jonathan Hayman

Computer Laboratory, University of Cambridge

Abstract. When defining the semantics of shared-memory concurrent programming languages, one conventionally has to make assumptions about the atomicity of actions such as assignments. Running on physical hardware, these assumptions can fail to hold in practice, which puts in question reasoning about their concurrent execution. We address an observation, due to John Reynolds, that processes proved sound in concurrent separation logic are separated to an extent that these assumptions can be disregarded, so judgements remain sound even if the assumptions on atomicity fail to hold. We make use of a Petri-net based semantics for concurrent separation logic with explicit representations of the key notions of ownership and interference. A new characterization of the separation of processes is given and is shown to be stronger than existing race-freedom results for the logic. Exploiting this, sufficient criteria are then established for an operation of refinement of processes capable of changing the atomicity of assignments.

1 Introduction

When giving the semantics of concurrent shared-memory programming languages, one has to choose a level of *granularity* for primitive actions. For example, an assignment $x := [y] + [y]$ which assigns to x twice the value held in the location pointed-to by y might be considered to be a primitive action that occurs in one step of execution. On the other hand, depending on the compiler and the system architecture, the assignment might be split in two, in effect running $x := [y]; x := x + [y]$. In a sequential setting, this is of little significance, but in a concurrent setting the second interpretation can give rise to additional behaviour. Suppose, for example, that the variable z points to the same heap location, ℓ say, as y; we say that y and z are *aliases*. If we run the programs above in parallel with the program $[z] := [z] + 1$ which increments the value at ℓ, it can be seen that $(x := [y] + [y]) \parallel ([z] := [z] + 1)$ always yields an even value in x, but $(x := [y]; x := x + [y]) \parallel ([z] := [z] + 1)$ terminates with an odd value in x if the assignment $[z] := [z] + 1$ occurs after $x := [y]$ but before $x := x + [y]$.

The key feature of the example above is that there is a *data-race*, *i.e.* an attempt to concurrently access the memory location ℓ. One of the key properties enforced by concurrent separation logic [8] is that every parallel process owns a disjoint region of memory and each process only accesses locations that it owns, from which it follows that proved processes are race-free. In [10], John Reynolds argues that race-freedom as enforced by the logic means that issues of granularity

can be disregarded; this has come to be known as *Reynolds' conjecture*. Due to the possibility of aliasing and the vital richness of the logic in allowing the transfer of ownership between processes, soundness of the logic and race-freedom are certainly non-trivial to prove, and led to the pioneering proof by Brookes [2] based on action traces. However, the intricacy of the powerful trace-based model stymied attempts to provide a formal proof of Reynolds' conjecture.

With the general goal of connecting concurrent separation logic and independence models for concurrency, in [7, 6] it was shown how to define a Petri-net based semantics for programming languages and, based on this, a semantics for the logic was developed. A key feature of independence models such as Petri nets is that they support notions of *refinement*: semantic operations to allow the provision of more accurate specifications of actions without affecting overall system behaviour [5]. General properties that are intrinsically linked to the independence of events connect the refined process back to the original process. In [7, 6], a form of refinement was given that could be applied to change the granularity assumed in the net semantics, and examples of its use were given. However, this did not fully address Reynolds' conjecture: there was no formal proof that the constraints governing when the refinement operation could be applied were always met for proved processes. This is tackled in this paper by establishing conditions on subprocesses and their refinements based on their *footprints*. In doing so, we reveal a new, stronger characterization of race-freedom arising from separation logic, which has the interesting side-effect of eliminating certain key examples of incompleteness of the logic.

Recently, Ferreira *et al.* have developed a new 'parameterized' operational semantics for programs and used it to show that a wide range of relaxations of assumptions of the memory model, including granularity, do not introduce extra behaviour for race-free processes [4]. However, not only does the net semantics presented here provide a valuable alternative perspective, with its direct account of ownership inherited from [7, 6], but it also shows how the Petri-net model directly supports reasoning about issues such as granularity: there is no need to extend the model, refinement being a native operation on nets, and so there is no need to provide a new proof of soundness of the logic as there is in [4]. This is part of a general programme of research aimed at demonstrating the use of independence models for concurrency, such as Petri nets, in the semantics of programming languages and showing how they natively support the study of a wide range of important aspects such as race-freedom, separation, granularity, weak memory models and memory-optimisation.

2 Syntax

In this section, we present the syntax of the programming language to be considered, and in the following section we present its Petri-net semantics. These sections are as in [6, 7] with the exception of equivalence at the end of Section 2 and Lemmas 1 and 2 in Section 3.

The programs that we consider operate on a *heap*, a finite partial map from a subset of heap locations Loc to values Val. We reserve the symbols $\ell, m, n, \ell', \ldots$ to range over heap locations and $h, h', \ldots$ to range over heaps. In a heap, a location can either be allocated, in which case the partial function is defined at that location, or be unallocated, in which case the partial function is undefined.

$$\text{Heap} = \text{Loc} \rightharpoonup_{\text{fin}} \text{Val}$$

Heap locations can point to other heap locations, so we have $\text{Loc} \subseteq \text{Val}$.

We now introduce a simple language for concurrent heap-manipulating programs. Its terms, ranged over by t, follow the grammar

$$\begin{array}{rcl} t & ::= & \alpha \mid \texttt{alloc}(\ell) \mid \texttt{dealloc}(\ell) \mid t_1; t_2 \mid t_1 \parallel t_2 \mid \alpha_1.t_1 + \alpha_2.t_2 \\ & & \mid \texttt{while}\ b\ \texttt{do}\ t\ \texttt{od} \mid \texttt{with}\ r\ \texttt{do}\ t\ \texttt{od} \end{array}$$

where α ranges over *heap actions*, b ranges over *Boolean guards* and r ranges over *resources*.

Heap actions represent arbitrary forms of action on the heap that neither allocate nor deallocate locations. For every action α, we assume that we are given a set

$$\mathcal{A}[\![\alpha]\!] \subseteq \text{Heap} \times \text{Heap}$$

such that $(h_1, h_2) \in \mathcal{A}[\![\alpha]\!]$ implies $\text{dom}(h_1) = \text{dom}(h_2)$. The set $\mathcal{A}[\![\alpha]\!]$ represents all the ways in which the action α can behave; the interpretation is that α can occur in a heap h if there exists $(h_1, h_2) \in \mathcal{A}[\![\alpha]\!]$ s.t. (the graph of) h_1 is contained in h, yielding a new heap in which the values held at locations in $\text{dom}(h_1)$ are updated according to h_2. For example, the semantics of an assignment $\ell\,\texttt{:=}\,v'$ is

$$\mathcal{A}[\![\ell\,\texttt{:=}\,v']\!] = \{(\{(\ell, v)\}, \{(\ell, v')\}) \mid v \in Val\}.$$

For any initial value v of ℓ, the heap can be updated to v' at ℓ. Further examples of action, such as pointer manipulation, are presented in [7, 6].

Boolean guards are heap actions that can occur only if the property that they represent holds. A full logic is given in [7, 6], but examples include:

- $\texttt{false}$, the action that can never occur (so $\mathcal{A}[\![\texttt{false}]\!] = \emptyset$),
- $\texttt{true}$, the action that can always occur (so $\mathcal{A}[\![\texttt{true}]\!] = \{(\emptyset, \emptyset)\}$),
- $\ell\,\texttt{?=}\,v$, the action that proceeds only if ℓ holds value v, and
- $[\ell]\,\texttt{!=}\,v$, the action that proceeds only if ℓ points to some location m that does not hold value v.

Allocation of heap locations can only occur through the term $\texttt{alloc}(\ell)$, which allocates an unused location and makes the existing location ℓ point to it, and deallocation can only occur through the term $\texttt{dealloc}(\ell)$, which deallocates the location pointed at by ℓ. There are primitives for iteration, sequential composition and parallel composition, and there is a form of guarded sum $\alpha_1.t_1 + \alpha_2.t_2$ which is a process that can run the process t_1 if the action α_1 can occur and can run the process t_2 if the action α_2 can occur. We sometimes use the notation $\texttt{if}\ b\ \texttt{then}\ t_1\ \texttt{endif}$ for the term $b.t_1 + \neg b.\texttt{true}$.

Finally, critical regions can be specified through the construct `with` r `do` t `od` indexed by *resources* r drawn from a set Res. This construct enforces the property that no two processes can concurrently be inside a critical region protected by the same resource r. This is implemented by recording a set of available resources; a critical region protected by r may only be entered if r is available, and whilst the process is in the critical region, the resource is marked as unavailable.

We define equivalence on terms to be the least congruence such that:

$$(t_1; t_2); t_3 \equiv t_1; (t_2; t_3) \qquad (t_1 \parallel t_2) \parallel t_3 \equiv (t_1 \parallel t_2) \parallel t_3 \qquad t_1 \parallel t_2 \equiv t_2 \parallel t_1.$$

A key tool in what follows shall be *term contexts*, terms with a single 'hole' denoted $-$. We use the symbols $k, k', \ldots$ to range over term contexts.

$$\begin{aligned} k \ ::= \quad & - \mid k; t \mid t; k \mid k \parallel t \mid t \parallel k \mid \alpha_1.k + \alpha_2.t \mid \alpha_1.t + \alpha_2.k \\ & \mid \texttt{while}\ b\ \texttt{do}\ k\ \texttt{od} \mid \texttt{with}\ r\ \texttt{do}\ k\ \texttt{od} \end{aligned}$$

We denote by $k[t]$ the term obtained by substituting the term t for the hole in k. Term contexts and equivalence will be used together to discuss the subprocesses of terms; for example, $\alpha; \beta$ is a subprocess of $\alpha; (\beta; \gamma)$ since there is a context, namely $-; \gamma$, such that $\alpha; (\beta; \gamma) \equiv (-; \gamma)[\alpha; \beta]$. Note the essential rôle here of equivalence as opposed to syntactic equality.

3 Petri-net semantics

Petri nets represent the behaviour of processes as collections of events that affect regions of local state called conditions. The particular form of net that we give semantics over is nets without multiplicity in which contact inhibits the occurrence of events — cf. the 'basic' nets of [12].

We place some additional structure on nets to form what we call *embedded nets*, which shall be used to define the semantics of programs. An embedded net is a tuple $(\mathbf{C}, \mathbf{S}, E, pre, post, I, T)$, where $\mathbf{C}$ is the set of *control* conditions, $\mathbf{S}$ is the set of *state* conditions, disjoint from $\mathbf{C}$, and $I, T \subseteq \mathbf{C}$ are the *initial* and *terminal* control conditions, respectively. We require that $(\mathbf{C} \cup \mathbf{S}, E, pre, post)$ forms a Petri-net with pre- and post-condition maps pre and $post$. As such, embedded nets are Petri nets equipped with a partition of their conditions into control and state conditions, alongside subsets of control conditions to indicate the initial and terminal control states of the process.

Any marking M of an embedded net can correspondingly be partitioned into (c, s) where $c = M \cap \mathbf{C}$ and $s = M \cap \mathbf{S}$. We write ${}^{\mathbf{C}}e$ for the control conditions that are preconditions to an event e, namely $pre(e) \cap C$, and define notation for the post-control conditions $e^{\mathbf{C}}$ and pre- and post-state conditions ${}^{\mathbf{S}}e$ and $e^{\mathbf{S}}$ similarly. We say that two embedded nets are *isomorphic* if there exists a bijection between their conditions and events that preserves the pre- and postconditions of events and preserves initial and terminal control markings. An isomorphism is said to be *state-preserving* if the restriction of the bijection to state conditions is the identity.

	Event e	${}^{\mathbf{S}}e$	$e^{\mathbf{S}}$
Heap action	$\mathsf{act}_{(c,c')}(h,h')$	h	h'
Allocation	$\mathsf{alloc}_{(c,c')}(\ell,v,\ell',v')$	$\{(\ell,v)\}$	$\{(\ell,\ell'),(\ell',v'),\mathsf{curr}(\ell')\}$
Deallocation	$\mathsf{dealloc}_{(c,c')}(\ell,\ell',v')$	$\{(\ell,\ell'),(\ell',v'),\mathsf{curr}(\ell')\}$	$\{(\ell,\ell')\}$
Enter CR	$\mathsf{acq}_{(c,c')}(r)$	$\{r\}$	$\emptyset$
Leave CR	$\mathsf{rel}_{(c,c')}(r)$	$\emptyset$	$\{r\}$

For all the above notations, ${}^{\mathbf{C}}e = c$ *and* $e^{\mathbf{C}} = c'$.

Table 1. Event notations

The embedded net representing a term t is denoted $\mathcal{N}[\![t]\!]$ with initial control conditions $\mathrm{Ic}(t)$ and terminal control conditions $\mathrm{Tc}(t)$. The inductive definition is presented in [7, 6]. The sets of control and state conditions are defined as follows:

Definition 1. *The control conditions* $\mathbf{C}$ *and state conditions* $\mathbf{S}$ *are defined as:*

- $\mathbf{C}$ *is ranged-over by* a *and follows the grammar*

$$a, a' ::= \mathsf{i} \mid \mathsf{t} \mid 1{:}a \mid 2{:}a \mid (a, a')$$

- $\mathbf{S} = \mathrm{Res} \cup (\mathrm{Loc} \times \mathrm{Val}) \cup \{\mathsf{curr}(\ell) \mid \ell \in \mathrm{Loc}\}$

A marking of state conditions $s \subseteq \mathbf{S}$ has $r \in s$ if the resource r is available; the heap holds value v at ℓ if $(l, v) \in s$; and the location ℓ has been allocated if $\mathsf{curr}(\ell) \in s$. Only certain markings of state conditions are sensible, namely those that are finite, satisfying $\mathsf{curr}(\ell) \in s$ iff there exists v s.t. $(\ell, v) \in s$, and if $(\ell, v), (\ell, v') \in s$ then $v = v'$. We call such markings (state) *consistent*. Given a consistent marking of state conditions s, we denote by $\mathsf{hp}(s)$ the heap in s, *i.e.* (the graph of) a partial function with finite domain:

$$\mathsf{hp}(s) = \{(\ell, v) \mid (\ell, v) \in s\}$$

Notations for the kinds of event that might be present in the net $\mathcal{N}[\![t]\!]$ are given in Table 1. The set of events for any process shall be extensional: any event is fully described just by its sets of pre- and postconditions. Two operations on events viewed in this way will be of particular use. The first prefixes a 'tag' onto the control conditions of an event. For any event e, the event $1 : e$ (and similarly $2{:}e$) is defined to have exactly the same effect as e on state conditions, ${}^{\mathbf{S}}(1{:}e) = {}^{\mathbf{S}}e$ and $(1{:}e)^{\mathbf{S}} = e^{\mathbf{S}}$, but using the tagged control conditions:

$${}^{\mathbf{C}}(1{:}e) = \{1{:}a \mid a \in {}^{\mathbf{C}}e\} \qquad (1{:}e)^{\mathbf{C}} = \{1{:}a \mid a \in e^{\mathbf{C}}\}$$

The second operation is used to 'glue' two nets together, for example forming $\mathcal{N}[\![t_1; t_2]\!]$ by pairing initial conditions of $\mathcal{N}[\![t_2]\!]$ with terminal conditions of $\mathcal{N}[\![t_1]\!]$. Given a set of control conditions $c \subseteq \mathbf{C}$ and a set $P \subseteq \mathbf{C} \times \mathbf{C}$, define

$$P \triangleleft c = \{(a, x) \in P \mid a \in c\} \cup \{a \in c \mid \nexists x.(a, x) \in P\}$$
$$P \triangleright c = \{(x, a) \in P \mid a \in c\} \cup \{a \in c \mid \nexists x.(x, a) \in P\}$$

This notation is applied to events e, yielding events $P \triangleleft e$ and $P \triangleright e$ with

$${}^{\mathbf{S}}(P \triangleleft e) = {}^{\mathbf{S}}(P \triangleright e) = {}^{\mathbf{S}}e \qquad ({}^{\mathbf{S}}P \triangleleft e) = ({}^{\mathbf{S}}P \triangleright e) = e^{\mathbf{S}}$$
$${}^{\mathbf{C}}(P \triangleleft e) = P \triangleleft ({}^{\mathbf{C}}e) \qquad (P \triangleleft e)^{\mathbf{C}} = P \triangleleft (e^{\mathbf{C}})$$
$${}^{\mathbf{C}}(P \triangleright e) = P \triangleright ({}^{\mathbf{C}}e) \qquad (P \triangleright e)^{\mathbf{C}} = P \triangleright (e^{\mathbf{C}})$$

The definitions of tagging and gluing extend to sets of events in the obvious way (for example, $1:E = \{1:e \mid e \in E\}$).

3.1 Net contexts and substitution

The net semantics for terms can be extended to give a net semantics for term contexts, giving what we call a *net context*. A net context simply specifies a control-point at which an embedded net may be placed.

Definition 2. *A* net context *is an embedded net with a distinguished event denoted* $[-]$ *such that* ${}^{\mathtt{s}}[-] = \emptyset = [-]^{\mathtt{s}}$.

The inductive definition of the net semantics of terms is extended in the obvious way to define a semantics for term contexts, denoted $\mathcal{N}[\![k]\!]$: the interpretation of the term context $-$ is the net context $\mathcal{N}[\![-]\!]$ with a single event $[-]$ with $pre([-]) = \{\mathtt{i}\}$ and $post([-]) = \{\mathtt{t}\}$.

Given a net context K and an embedded net N, we now define an embedded net representing the substitution of N for the hole $[-]$ in K. This is simply obtained by using tagging to force the events of K and N to be disjoint, removing the artificial 'hole' event from K and then 'gluing' N in the appropriate place.

Definition 3. *Let the embedded net* $N = (\mathbf{C}, \mathbf{S}, E_N, pre_N, post_N, I_N, T_N)$ *and let the net context* $K = (\mathbf{C}, \mathbf{S}, E_K, pre_K, post_K, I_K, T_K)$. *Define the subsets of control conditions*

$$P_{\mathsf{init}} = 1:pre_K([-]) \times 2:I_N \qquad P_{\mathsf{term}} = 1:post_K([-]) \times 2:T_N.$$

The embedded net $K[N] = (\mathbf{C}, \mathbf{S}, E, pre, post, I, T)$ *is defined as*

$$I = (P_{\mathsf{init}} \cup P_{\mathsf{term}}) \lhd 1:I_K \qquad T = (P_{\mathsf{init}} \cup P_{\mathsf{term}}) \lhd 1:T_K$$
$$E = (P_{\mathsf{init}} \cup P_{\mathsf{term}}) \lhd (1:(E_K \setminus \{[-]\})) \cup (P_{\mathsf{init}} \cup P_{\mathsf{term}}) \rhd 2:E_N$$

An example is shown in Figure 1 (eliding the unaffected state conditions). Note that the conditions $P_{\mathsf{init}} = \{(1:c, 2:w), (1:c, 2:x), (1:d, 2:w), (1:d, 2:x)\}$ are marked when the initial control point of N is reached. Generally, we say that the subprocess N in $K[N]$ is *initialized* in the marking of control conditions c when $P_{\mathsf{init}} \subseteq c$.

Net contexts and net substitution connect with term contexts and substitution through the following lemma.

Lemma 1. *There is a state-preserving isomorphism of embedded nets*

$$\gamma_{k,t} : \mathcal{N}[\![k[t]]\!] \cong \mathcal{N}[\![k]\!]\,[\mathcal{N}[\![t]]\!].$$

It now becomes possible to determine when a subprocess t in $k[t]$ becomes active. Given a marking c of control conditions in $\mathcal{N}[\![k[t]]\!]$, let $\gamma_{k,t}c$ denote the subset of control conditions of $\mathcal{N}[\![k]\!]\,[\mathcal{N}[\![t]]\!]$ obtained as the image of the set c under $\gamma_{k,t}$. We shall say that t *is initialized in* c if, under the isomorphism, the set of conditions P_{init} is marked; that is, if $P_{\mathsf{init}} \subseteq \gamma_{k,t}c$. The semantics of terms ensures that if t is initialized in c, no condition inside t is marked and no condition corresponding to a terminal condition of t is marked.

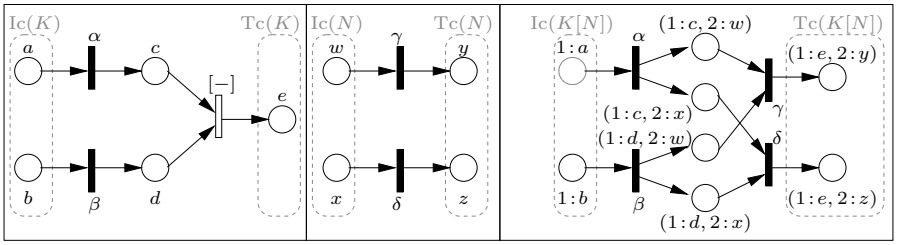

Fig. 1. Application of context K to net N, yielding $K[N]$.

$$\frac{\begin{array}{r}\forall h \models \varphi : \forall (h_1, h_2) \in \mathcal{A}[\![\alpha]\!] : \\ \forall h' \supseteq h : (h_1 \subseteq h' \implies h_1 \subseteq h) \\ \&\ (h_1 \subseteq h \implies (h \setminus h_1) \cup h_2 \models \psi)\end{array}}{\Gamma \vdash \{\varphi\}\alpha\{\psi\}} \qquad \frac{\Gamma \vdash \{\varphi_1\}t_1\{\psi_1\} \quad \Gamma \vdash \{\varphi_2\}t_2\{\psi_2\}}{\Gamma \vdash \{\varphi_1 \star \varphi_2\}t_1 \parallel t_2\{\psi_1 \star \psi_2\}}$$

$$\frac{\Gamma, w : \chi \vdash \{\varphi \star \chi\}t\{\psi \star \chi\}}{\Gamma, w : \chi \vdash \{\varphi\}\texttt{with}\ w\ \texttt{do}\ t\ \texttt{od}\{\psi\}}$$

Fig. 2. Selected rules of concurrent separation logic

Lemma 2. *For any initial marking* $(\mathrm{Ic}(k[t]), s_0)$ *of* $\mathcal{N}[\![k[t]]\!]$*, if* (c, s) *is reachable and* $\gamma_{k,t}^{-1} P_{\mathsf{init}} \subseteq c$ *then* $c = \gamma_{k,t}^{-1} P_{\mathsf{init}} \cup 1 : c_1$ *for some set of control conditions* c_1*.*

The property is shown by consideration of the *control nets* described in [7, 6].

4 Concurrent separation logic

Concurrent separation logic is a Hoare-style system designed to provide partial correctness judgements about concurrent heap-manipulating programs. We refer the reader to [8] for a full introduction to the logic.[1]

In this section, we briefly summarize the essential parts of the Petri-net model presented in [7, 6], to which we refer the reader for the full definition of the syntax and semantics of the logic. A selection of rules is presented in Figure 2. The key judgement is $\Gamma \vdash \{\varphi\}t\{\psi\}$, which has the following interpretation:

> *If initially* φ *holds of the heap defined at the locations* owned *by the process, then, after* t *runs to completion,* ψ *holds of the heap defined at the locations owned by the process; during any such run, the process only accesses locations that it owns and preserves* invariants *in* Γ*.*

At the core of separation logic is the separating conjunction, $\varphi_1 \star \varphi_2$. A heap h satisfies this formula, written $h \models \varphi_1 \star \varphi_2$, if h can be partitioned into disjoint subheaps h_1 and h_2 such that $h_1 \models \varphi_1$ and $h_2 \models \varphi_2$.

[1] Here, we make two simplifications to the logic that are orthogonal to our results: we do not distinguish stack and heap variables and we do not give a rule for declaration of new resources (so the environment Γ is fixed through any derivation).

Abbreviation	Preconditions	Postconditions
$\overline{\mathsf{act}}(h_1, h_2)$	$h_1 \cup \{\omega_{\mathsf{oth}}(\ell) \mid \ell \in \mathrm{dom}(h_1)\}$	$h_2 \cup \{\omega_{\mathsf{oth}}(\ell) \mid \ell \in \mathrm{dom}(h_1)\}$
$\overline{\mathsf{alloc}}(\ell, v, \ell', v')$	$\{\omega_{\mathsf{oth}}(\ell), (\ell, v)\}$	$\{\omega_{\mathsf{oth}}(\ell), \omega_{\mathsf{oth}}(\ell'), \mathsf{curr}(\ell'), (\ell, \ell'), (\ell', v')\}$
$\overline{\mathsf{dealloc}}(\ell, \ell', v')$	$\{\omega_{\mathsf{oth}}(\ell), \omega_{\mathsf{oth}}(\ell'), \mathsf{curr}(\ell'), (\ell, \ell'), (\ell', v')\}$	$\{\omega_{\mathsf{oth}}(\ell), (\ell, \ell')\}$
$\overline{\mathsf{acq}}(r, h)$	$\{\omega_{\mathsf{inv}}(r), r\} \cup h \cup \{\omega_{\mathsf{inv}}(\ell) \mid \exists v.(\ell, v) \in h\}$	$\{\omega_{\mathsf{oth}}(r)\} \cup h \cup \{\omega_{\mathsf{oth}}(\ell) \mid \exists v.(\ell, v) \in h\}$
$\overline{\mathsf{rel}}(r, h)$	$\{\omega_{\mathsf{oth}}(r)\} \cup h \cup \{\omega_{\mathsf{oth}}(\ell) \mid \exists v.(\ell, v) \in h\}$	$\{\omega_{\mathsf{inv}}(r), r\} \cup h \cup \{\omega_{\mathsf{inv}}(\ell) \mid \exists v.(\ell, v) \in h\}$

Table 2. Interference events

The *environment* Γ associates an *invariant* to every resource free in t. An invariant is a *precise* heap formula: a formula χ is said to be precise if given any heap h, there is at most one heap $h_0 \subseteq h$ such that $h_0 \models \chi$. When a process enters a critical region, it gains ownership of the part of the heap that satisfies the invariant. When it leaves the critical region, it is required to have restored the invariant and it loses ownership of the associated part of the heap. This is reflected in the rule for critical regions. As seen in [8], ownership of locations can be transferred between processes using critical regions. This provides vital power to the logic but introduces subtlety to the notion of ownership since the set of locations that the process owns changes as it executes.

4.1 Interference and ownership nets

In order to demonstrate the correctness of the rule for parallel composition, we shall reason about the process running in the presence of arbitrary processes that act only on locations that they are seen to own. Central to this interpretation is a formal treatment of ownership, in which locations are partitioned into three disjoint sets: locations that are owned by the process, locations that are used to satisfy the invariants of available resources and locations that are owned by other processes.

The first stage in the creation of the formal model is the definition of an *interference net*, a net to simulate the behaviour of the arbitrary concurrently-executing processes on the shared state. The constraints on their behaviour are represented through the presence of explicit conditions to represent ownership in the system. Notation to describe the kinds of event present in the interference net is given in Table 2.

Definition 4. *The set of* ownership conditions *is defined to be the set* $\mathbf{W} = \{\omega_{\mathsf{proc}}(x), \omega_{\mathsf{inv}}(x), \omega_{\mathsf{oth}}(x) \mid x \in \mathrm{Loc} \cup \mathrm{Res}\}$. *The* interference net *for* Γ *has conditions* $\mathbf{S} \cup \mathbf{W}$ *and events called* interference events*:*

- $\overline{\mathsf{act}}(h_1, h_2)$ *for all heaps* h_1 *and* h_2 *such that* $\mathrm{dom}(h_1) = \mathrm{dom}(h_2)$
- $\overline{\mathsf{alloc}}(\ell, v, \ell', v')$ *and* $\overline{\mathsf{dealloc}}(\ell, \ell', v')$ *for all* ℓ *and* ℓ' *and values* v *and* v'
- $\overline{\mathsf{acq}}(r, h)$ *and* $\overline{\mathsf{rel}}(r, h)$ *for all* $r \in \mathrm{dom}(\Gamma)$ *and* h *such that* $h \models \chi$*, for* χ *the unique formula such that* $r : \chi \in \Gamma$

We use the symbol u to range over interference events and use w to range over markings of ownership conditions. A marking $\sigma = (s, w)$ of the interference net is said to be *consistent* if σ is consistent and, for each $z \in \mathrm{Loc} \cup \mathrm{Res}$, if either $z \in \mathrm{Res}$ or $\mathsf{curr}(z) \in s$ then there is precisely one condition in

$\{\omega_{\mathsf{proc}}(z), \omega_{\mathsf{inv}}(z), \omega_{\mathsf{oth}}(z)\} \cap w$; otherwise, if $z \in \mathrm{Loc}$ and $\mathsf{curr}(z) \notin s$, we require that the set $\{\omega_{\mathsf{proc}}(z), \omega_{\mathsf{inv}}(z), \omega_{\mathsf{oth}}(z)\} \cap w$ be empty. As such, a consistent marking assigns precisely one ownership state to every current location and resource.

The interference net for an environment Γ describes the potential behaviour of other processes that can take place on the state. For example, the interference event $\overline{\mathsf{act}}(h_1, h_2)$ can update the heap only if the locations operated-on are seen as owned by 'other' processes. The interpretation of a judgement $\Gamma \vdash \{\varphi\}t\{\psi\}$ shall therefore consider the net for t running in parallel with the interference net. However, additionally, the symmetry in the rule for parallel composition requires that the behaviour of t can be seen as interference when considering other processes; we establish this by *synchronization* of $\mathcal{N}[\![t]\!]$ with the interference net for Γ. We begin by defining with which interference events an event e of $\mathcal{N}[\![t]\!]$ can synchronize:

- the event $\mathsf{act}_{(c,c')}(h_1, h_2)$ can synchronize with $\overline{\mathsf{act}}(h_1, h_2)$,
- the event $\mathsf{alloc}_{(c,c')}(\ell, v, \ell', v')$ can synchronize with $\overline{\mathsf{alloc}}(\ell, v, \ell', v')$,
- the event $\mathsf{dealloc}_{(c,c')}(\ell, \ell', v')$ can synchronize with $\overline{\mathsf{dealloc}}(\ell, \ell', v')$,
- the event $\mathsf{acq}_{(c,c')}(r)$ can synchronize with $\overline{\mathsf{acq}}(r, h)$ for any h, and
- the event $\mathsf{rel}_{(c,c')}(r)$ can synchronize with $\overline{\mathsf{rel}}(r, h)$ for any h.

Suppose that two events synchronize, e from the process and u from the interference net. The event u is the event that would fire in the net for the other parallel process to simulate the event e. Let $e \cdot u$ be the event formed by taking the union of the preconditions (and, respectively, postconditions) of e and u, other than using $\omega_{\mathsf{proc}}(\ell)$ in place of $\omega_{\mathsf{oth}}(\ell)$, and similarly $\omega_{\mathsf{proc}}(r)$ in place of $\omega_{\mathsf{oth}}(r)$.

Definition 5. *The* ownership net $\mathcal{W}[\![t]\!]_\Gamma$ *is the net with conditions* $\mathbf{C} \cup \mathbf{S} \cup \mathbf{W}$ *and all events that are either events* u *from the interference net for* Γ *or synchronized events* $e \cdot u$ *where* e *is an event of* $\mathcal{N}[\![t]\!]$ *and* u *is an interference event such that* e *and* u *can synchronize.*

Markings of ownership nets are tuples (c, s, w) where c, s and w are the markings of, respectively, control, state and ownership conditions. We say that (c, s, w) is consistent if (s, w) is consistent. It is shown in [7, 6] that the property of markings being consistent is preserved as processes execute.

4.2 Soundness

The formulation of the ownership net permits a fundamental understanding of when a process acts in a way that would not be seen as interference when considering other processes, by failing to respect ownership or to restore invariants.

Definition 6 (Violating marking). *We say that a consistent marking* (c, s, w) *of* $\mathcal{W}[\![t]\!]_\Gamma$ *is* violating *if there exists an event* e *of* $\mathcal{N}[\![t]\!]$ *that has concession in the marking* (c, s) *but there is no event* u *from the interference net such that* u *synchronizes with* e *and* $e \cdot u$ *has concession in* (c, s, w).

We are now ready to turn to soundness of judgements. Given a state s and environment Γ, let $\mathrm{inv}(\Gamma, s)$ denote the separating conjunction of formulae χ_r s.t. $r \in R$ and $r : \chi_r \in \Gamma$; so $\mathrm{inv}(\Gamma, s)$ is a formula that is satisfied by a heap that can be split into separate parts, each of which satisfies the invariant for a distinct available resource.

Definition 7. *Let s be a state containing heap h. For any $L \subseteq \mathrm{Loc}$, let $h \restriction L$ denote the restriction of h to L, so $h \restriction L = \{(\ell, v) \mid \ell \in L \text{ and } v \in \mathrm{Val} \text{ and } (\ell, v) \in s\}$. The marking (c, s, w) is said to* satisfy φ in Γ *if (c, s, w) is consistent, $h \restriction \{\ell \mid \omega_{\mathsf{inv}}(\ell) \in w\} \models \mathrm{inv}(\Gamma, s)$, and $h \restriction \{\ell \mid \omega_{\mathsf{proc}}(\ell) \in w\} \models \varphi$.*

We now present soundness of the system: see [7, 6] for a proof and also formal results connecting this down to the behaviour of the original process $\mathcal{N}[\![t]\!]$.

Theorem 1. *If $\Gamma \vdash \{\varphi\} t \{\psi\}$ then, for any s and w such that $(\mathrm{Ic}(t), s, w)$ satisfies φ in Γ, no violating marking is reachable from $(\mathrm{Ic}(t), s, w)$ in $\mathcal{W}[\![t]\!]_\Gamma$ and if $(\mathrm{Tc}(t), s', w')$ is reachable from $(\mathrm{Ic}(t), s, w)$ then $(\mathrm{Tc}(t), s', w')$ satisfies ψ in Γ.*

5 Separation

In this section, we use the ownership semantics described above to capture how the subprocesses of any proved term can be separated from their environment. In [7, 6], the characterization was based solely on independence of events; here, we provide a stronger result based on ownership.

First, we extend the construction of the ownership net to contexts, yielding ownership nets $\mathcal{W}[\![k]\!]_\Gamma$ consisting of events that are either interference events from the interference net for Γ, synchronized events as described above or the hole event $[-]$ drawn from $\mathcal{N}[\![k]\!]$. Note that Lemma 2 extends straightforwardly to ownership nets: Given an initial marking $(\mathrm{Ic}(k[t]), s_0, w_0)$ of an ownership net $\mathcal{W}[\![k[t]]\!]_\Gamma$, any reachable marking (c, s, w) such that t is initialized in c satisfies $c = \gamma_{k,t}^{-1} P_{\mathsf{init}} \cup 1 : c_1$ for some (necessarily unique) subset of control conditions c_1.

Central to characterizing how a term t and context k can be separated is the ability to split their ownership. Let w, w_1 and w_2 be markings of ownership conditions. Then w_1 and w_2 form an *ownership split* of w if for all $z \in \mathrm{Loc} \cup \mathrm{Res}$:

$$\begin{gathered}
\omega_{\mathsf{proc}}(z) \in w \iff \omega_{\mathsf{proc}}(z) \in w_1 \cup w_2 \\
\omega_{\mathsf{oth}}(z) \in w \iff \omega_{\mathsf{oth}}(z) \in w_1 \cap w_2 \\
\omega_{\mathsf{inv}}(z) \in w \iff \omega_{\mathsf{inv}}(z) \in w_1 \iff \omega_{\mathsf{inv}}(z) \in w_2
\end{gathered}$$

Definition 8 (Separability and subprocess race-freedom). *With respect to an environment Γ, say that k and t are* separable *from (s_0, w_0) if, for any marking (c, s, w) reachable from $(\mathrm{Ic}(k[t]), s_0, w_0)$ in $\mathcal{W}[\![k[t]]\!]_\Gamma$ such that t is initialized in c, there exist w_1 and w_2 forming an ownership split of w satisfying*

- *no violating marking is reachable from (c_1, s, w_1) in $\mathcal{W}[\![k]\!]_\Gamma$ by events excluding $[-]$, where c_1 is the set such that $c = \gamma_{k,t}^{-1} P_{\mathsf{init}} \cup 1 : c_1$, and*
- *no violating marking is reachable from $(\mathrm{Ic}(t), s, w_2)$ in $\mathcal{W}[\![t]\!]_\Gamma$.*

Say that a term t_0 is subprocess race-free *from (s_0, w_0) if, for all k and t such that $t_0 \equiv k[t]$, it is the case that k and t are separable from (s_0, w_0).*

Intuitively, if a marking is encountered in $\mathcal{W}[\![k[t]]\!]_\Gamma$ in which t is initialized, that k and t are separable means that the ownership of the heap and resources can be partitioned between t and k in such a way that the allocation of resources to t is sufficient that it never acts on anything that it does not own, and the allocation of resources to k is sufficient that any action that k can perform prior to the completion of t is constrained to be on the locations that k owns.

Theorem 2. *If $\Gamma \vdash \{\varphi\}t\{\psi\}$ then t_0 is subprocess race-free from any (s_0, w_0) that satisfies φ in Γ.*

5.1 Strength of race-freedom

Subprocess race-freedom is unusual in its use of ownership; as we shall see, this is intimately related to the logic and shall be central to our constraint on refinement. We first study its position in a hierarchy of race-freedom properties.

An interesting alternative candidate is to say that a term t_0 is *AAD race-free* from marking (s_0, w_0) if, for any context k and any term t that is either a heap action, an allocation or a deallocation command such that $t_0 \equiv k[t]$, then k and t are separable from (s_0, w_0). (A full treatment of this would extend contexts to allow the hole to occur at guards in loops and the sum.)

AAD race-freedom is a stronger property than the more familiar notions of race-freedom [2, 7, 6] which require that no two actions can occur concurrently on the same memory location. For example, consider the process t defined as $\ell := 0$ and context k defined as $- \parallel \mathtt{alloc}(m); \mathtt{dealloc}(m); \mathtt{if}\ m = \ell\ \mathtt{then}\ \ell := 1\ \mathtt{endif}$ (the Boolean $m = \ell$ passes only if m is a pointer to ℓ). From any initial state in which ℓ and m are owned by $k[t]$, it is easy to see that the allocation command can never allocate ℓ so there is never any concurrent access of any memory location. However, these processes are not AAD race-free. To see this, we would certainly have to give ownership of ℓ to the assignment $\ell := 0$. Consequently, in the net $\mathcal{W}[\![k]\!]_\emptyset$, an interference event could occur in which ℓ is deallocated followed by allocation of ℓ by the allocation command and subsequently by the assignment of 1 to the location ℓ which is not owned by the process.

Subprocess race-freedom is an even more discriminating condition than AAD race-freedom. Consider the net $\mathcal{W}[\![k'[t']]\!]_\emptyset$ where

$$\begin{aligned} t' = &\quad n := 0; \mathtt{dealloc}(p) \\ k' = &\quad - \parallel \mathtt{alloc}(\ell); \mathtt{while}\ (\ell\ \mathtt{!=}\ m)\ \mathtt{do}\ \mathtt{alloc}(\ell)\ \mathtt{od}; n := 1 \end{aligned}$$

running from an initial state with heap $\{(\ell, 0), (m, 0), (n, 0), (p, m)\}$ and ownership marking $\{\omega_{\mathsf{proc}}(\ell), \omega_{\mathsf{proc}}(m), \omega_{\mathsf{proc}}(n), \omega_{\mathsf{proc}}(p)\}$. Going through each action, it can be verified that $k'[t']$ is AAD race-free; the key is that $n := 1$ can only happen after $\mathtt{dealloc}(p)$. However, $k'[t']$ is not subprocess race-free: ownership of n, p and m must be given to t', which means that when considering the context

$\mathcal{W}[\![k']\!]_\emptyset$, it becomes possible for an interference event deallocating m to occur, then re-allocation of m by k' followed by assignment to the unowned location n.

Interestingly, the most important known examples of the incompleteness of concurrent separation logic all involve processes that are not subprocess race-free according to the definition here, so one may hope that this tighter form of race-freedom gives new insight towards (relative) completeness.

6 Footprints and refinement

We have seen that any context k and term t that form a proved process can be separated from suitable initial states. We now introduce an operation of refinement of t by some other process t'. Of course, this is only permitted when the interaction of t and t' with k is restricted; in particular, we shall restrict to processes that do not have critical regions or allocate or deallocate locations.

Definition 9. *A term z is* static *if it follows the grammar*

$$z ::= \alpha \mid z_1; z_2 \mid \alpha_1.z_1 + \alpha_2.z_2 \mid z_1 \parallel z_2 \mid \texttt{while}\ b\ \texttt{do}\ z\ \texttt{od}.$$

The key goal is to show that, for static terms z and z', if $k[z']$ runs from a suitable initial state to a terminal state, there should be a corresponding run of $k[z]$ from the initial state to the same terminal state.

Two constraints shall be necessary when considering whether a static term z' can replace z in a context k. The first is that if z' runs from an initial state s to a terminal state s' then z can also run from s to s'. For any term t, write $t : s \Downarrow s'$ if the marking $(\mathrm{Tc}(t), s')$ is reachable from $(\mathrm{Ic}(t), s)$ in $\mathcal{N}[\![t]\!]$. We shall require that if $z' : s \Downarrow s'$ then $z : s \Downarrow s'$.

The second constraint is that, running from any state s, the locations that z' accesses are all locations that z might access. To justify this, consider the following example. It is easy to see that there are no s and s' such that $\ell\,\texttt{?=}\,0; \ell\,\texttt{?=}\,1 : s \Downarrow s'$, so the first constraint for using $\ell\,\texttt{?=}\,0; \ell\,\texttt{?=}\,1$ to replace $\texttt{false}$ in the process $\texttt{false} \parallel \ell\,\texttt{:=}\,1$ would be met. However, the resulting process $\ell\,\texttt{?=}\,0; \ell\,\texttt{?=}\,1 \parallel \ell\,\texttt{:=}\,1$ has more behaviour, so the refinement is unsound. It should be ruled-out because the command $\texttt{false}$ accesses no locations whereas $\ell\,\texttt{?=}\,0; \ell\,\texttt{?=}\,1$ accesses ℓ.

The locations that might be accessed by z are called its *footprint*, which we now capture as the least allocation of ownership of the heap to the process that ensures that no violation is encountered. The notion of footprint has intricacies, but since our interest is in the footprint of static terms, we can avoid many of them. In particular, we do not need to consider ownership of invariants; we shall say that a marking of ownership conditions w is *invariant-empty* if there exists no $z \in \mathrm{Loc} \cup \mathrm{Res}$ such that $\omega_{\mathsf{inv}}(z) \in w$.

Let w and w' be invariant-empty markings of ownership conditions consistent with some state s. Define $w \leq w'$ if $\omega_{\mathsf{proc}}(z) \in w$ implies $\omega_{\mathsf{proc}}(z) \in w'$ for all $z \in \mathrm{Loc} \cup \mathrm{Res}$. For two invariant-empty ownership markings w and w' consistent with the state s, define

$$w \sqcap_s w' = \begin{array}{c} \{\omega_{\mathsf{proc}}(z) \mid \omega_{\mathsf{proc}}(z) \in w \text{ and } \omega_{\mathsf{proc}}(z) \in w'\} \\ \cup\{\,\omega_{\mathsf{oth}}(z) \mid \omega_{\mathsf{oth}}(z) \in w \text{ or } \omega_{\mathsf{oth}}(z) \in w'\,\}. \end{array}$$

It is easy to see that this is consistent and is a least upper bound of w and w' w.r.t. the partial order $\leq$ over invariant-empty ownership markings consistent with s.

Lemma 3. *Let z be a static term. For any s and invariant-empty w and w' such that both (s, w) and (s, w') are consistent, if no violating marking is reachable from either $(\mathrm{Ic}(z), s, w)$ or $(\mathrm{Ic}(z), s, w')$ in $\mathcal{W}[\![z]\!]_\emptyset$ then no violating marking is reachable from $(\mathrm{Ic}(z), s, w \sqcap_s w')$ in $\mathcal{W}[\![t]\!]_\emptyset$.*

Recalling that the marking s must be finite, it follows immediately from this lemma that, for any static term z and state s, there exists a *least* (according to the order $\leq$) invariant-empty marking of ownership conditions w consistent with s such that no violating marking is reachable in $\mathcal{W}[\![z]\!]_\emptyset$ from $(\mathrm{Ic}(z), s, w)$. The locations that must be owned by the process form the footprint of t:

$$\mathsf{footprint}(z, s) = \{\ell \mid \omega_{\mathsf{proc}}(\ell) \in w\}$$

The restriction to static terms in Lemma 3 is important: there are examples of non-static processes for which this property fails. For example, consider the state with heap $\{(k, 0), (l, 0), (m, 0)\}$ and term $\texttt{while}\ m\ \texttt{!=}\ k\ \texttt{do}\ \texttt{alloc}(m)\ \texttt{od}; \ell\ \texttt{:=}\ 1$. No violating marking is reachable from either of the initial ownership markings $\{\omega_{\mathsf{proc}}(m), \omega_{\mathsf{proc}}(\ell), \omega_{\mathsf{oth}}(k)\}$ or $\{\omega_{\mathsf{proc}}(m), \omega_{\mathsf{oth}}(\ell), \omega_{\mathsf{proc}}(k)\}$ but a violating marking is reachable from their l.u.b., $\{\omega_{\mathsf{proc}}(m), \omega_{\mathsf{oth}}(\ell), \omega_{\mathsf{oth}}(k)\}$.

We now give a key result, that footprint-respecting refinements give rise to no additional behaviour.

Theorem 3. *Let z and z' be static terms such that, for all states s and s':*

$$z' : s \Downarrow s' \implies z : s \Downarrow s' \qquad \textit{and} \qquad \mathsf{footprint}(z', s) \subseteq \mathsf{footprint}(z, s)$$

Let k be a context such that k and z are separable from (s_0, w_0). Then:

- *k and z' are separable from (s_0, w_0),*
- *if no violating marking is reachable in $\mathcal{W}[\![k[z]]\!]_\Gamma$ from (s_0, w_0) then no violating marking is reachable in $\mathcal{W}[\![k[z']]\!]_\Gamma$, and*
- *if the terminal marking $(\mathrm{Tc}(k[z']), s, w)$ is reachable from $(\mathrm{Ic}(k[z']), s_0, w_0)$ in $\mathcal{W}[\![k[z']]\!]_\Gamma$ then the terminal marking $(\mathrm{Tc}(k[z]), s, w)$ is reachable from $(\mathrm{Ic}(k[z]), s_0, w_0)$ in $\mathcal{W}[\![k[z]]\!]_\Gamma$.*

The proof proceeds similarly to that for 'non-interfering substitutions' in [7, 6], using the fact that any two consecutive occurrences of events whilst z' is active will be independent if one event is from k and one is from z'.

We now show how the property of being subprocess race-free is preserved under the forms of refinement described above.

Theorem 4. *Let z and z' be static terms such such that, for all states s and s':*

- *$z' : s \Downarrow s'$ implies $z : s \Downarrow s'$ and $\mathsf{footprint}(z', s) \subseteq \mathsf{footprint}(z, s)$, and*
- *for any w such that (s, w) is consistent, if z is subprocess race-free from (s, w) then z' is subprocess race-free from (s, w), both taking the environment to be empty.*

For any context k, environment Γ and consistent (s_0, w_0), if $k[z]$ is subprocess race-free from (s_0, w_0) then $k[z']$ is subprocess race-free from (s_0, w_0).

Together, Theorems 3 and 4 show how the validity of judgements is preserved by footprint-preserving refinements of static subprocesses.

We conclude by giving an example of the kind of refinement that is permitted, showing how any static subterm can be refined to its effect. For a static term z define the action $\mathsf{collapse}_z$ as

$$\mathcal{A}\,[\![\mathsf{collapse}_z]\!] \stackrel{\text{def}}{=} \left\{ (\mathsf{hp}(s), \mathsf{hp}(s')) \;\middle|\; \begin{array}{r} z : s \Downarrow s' \text{ and for all } s_0 \text{ s.t. } s \subseteq s_0 : \\ \mathsf{footprint}(z, s_0) = \mathrm{dom}(\mathsf{hp}(s)) \end{array} \right\}.$$

The action is formed of minimal heaps that represent the fault-avoiding (*i.e.* sufficient that z never accesses an unallocated location) big-step semantics of z. It is easy to see that the conditions for $\mathsf{collapse}_z$ replacing any static subterm z in any context k are met, so the obtained semantics is related to the original semantics by Theorems 3 and 4.

An adaptation of this would be to define two actions, one representing the start of z and the other the end of z. The events of the 'start' action record the part of the heap to be modified by z and the 'end' events would perform the update, yielding a semantics for proved processes following that in [10].

7 Conclusions and related work

We have seen how a Petri-net semantics for concurrent separation logic can be used to prove that its judgements are insensitive to the granularity assumed of primitive actions. In particular, through net and term contexts, an interpretation of the subprocesses of programs was introduced and it was shown how ownership can be split between any subprocess and its context in such a way that neither exceeds the constraints imposed by ownership. The issue of granularity was then addressed by showing that if the footprint of a refinement of a subprocess does not exceed the original footprint, the validity of judgements is preserved.

We have seen how refinements of static terms can begin to yield a semantics along the lines of Reynolds' model [10], with static terms being replaced by 'start' and 'end' actions. His model, however, treats races as 'catastrophic'; we directly prove that they cannot occur. This may be of use when considering refinements of separable parts of racy programs. Related to Reynolds' model is Brookes' footstep trace model [1], in which sequences of actions in individual traces are 'collapsed' to their effect. The goal of the footstep model is to move towards logical full abstraction. However, work presented there (and Reynolds') is critically different from that presented here in that no general connection is shown between the behaviour of processes following change in the granularity of actions and the original processes. The model in [1] also bypasses the important issue of interaction between concurrent processes through allocation or deallocation of memory; it assumes that all allocated memory locations are 'fresh', whereas in real implementations the opposite is often the case. Reynolds' model, on the other hand, has no allocation or deallocation at all.

As discussed in the introduction, in [4] it has recently been shown how a range of assumptions relating both to granularity and other aspects of 'relaxed' memory models can be ignored for programs proved in separation logic. The key difference between their model and this one is that the net-based semantics *natively* permits refinements, whereas their model relies on an additional 'parameterization' relation in the semantics. This leads to them having to re-prove concurrent separation logic sound. However, their constraints on their parameterization relations under which the validity of judgements is preserved are similar to those seen here, so it would be interesting to provide a full connection with their work, in particular to consider how the net model can be applied to prove sound the other forms of memory relaxation described there.

There are a number of areas worthy of further investigation, one of which is the extension of refinement beyond static terms. It seems as though a treatment of refinement of terms involving allocation and deallocation would require a more subtle interpretation of footprint, perhaps along the lines of that presented in [9]. More broadly, the net model here and the abstract semantics for concurrent separation logic [3] deserve connection, and thereon, for example, to RGSep [11].

Acknowledgements: I would like to thank Glynn Winskel, Matthew Parkinson and the anonymous referees for useful comments, and gratefully acknowledge the support of the ERC Advanced Grant ECSYM and the Leverhulme Trust.

References

1. Brookes, S.: A grainless semantics for parallel programs with shared mutable data. In: Proc. MFPS XXI. ENTCS (2005)
2. Brookes, S.: A semantics for concurrent separation logic. Theoretical Computer Science 375(1–3) (2007)
3. Calcagno, C., O'Hearn, P.W., Yang, H.: Local action and abstract separation logic. In: Proc. LICS 2007. IEEE Press (2007)
4. Ferreira, R., Feng, X., Shao, Z.: Parameterized memory models and concurrent separation logic. In: Proc. ESOP 2010. LNCS, vol. 6012. Springer (2010)
5. van Glabbeek, R.J., Goltz, U.: Equivalence notions for concurrent systems and refinement of actions. In: Proc. MFCS 1989. LNCS, vol. 379. Springer-Verlag (1989)
6. Hayman, J.M.: Petri net semantics. Ph.D. thesis, University of Cambridge, Computer Laboratory (2009), available as Technical Report UCAM-CL-TR-782.
7. Hayman, J.M., Winskel, G.: Independence and concurrent separation logic. Logical Methods in Computer Science 4(1) (2008), special issue for LICS '06
8. O'Hearn, P.W.: Resources, concurrency and local reasoning. Theoretical Computer Science 375(1–3), 271–307 (2007)
9. Raza, M., Gardner, P.: Footprints in local reasoning. Logical Methods in Computer Science 5(2), 1–27 (2009)
10. Reynolds, J.C.: Toward a grainless semantics for shared-variable concurrency. In: Proc. FSTTCS 2004. LNCS, vol. 3328. Springer-Verlag (2004)
11. Vafeiadis, V., Parkinson, M.: A marriage of Rely/Guarantee and separation logic. In: Proc. CONCUR 2007. LNCS, vol. 4703. Springer-Verlag (2007)
12. Winskel, G., Nielsen, M.: Models for concurrency. In: Handbook of Logic and the Foundations of Computer Science, vol. 4, pp. 1–148. Oxford University Press (1995)