Characterising Testing Preorders for Broadcasting Distributed Systems (Extended Version)*

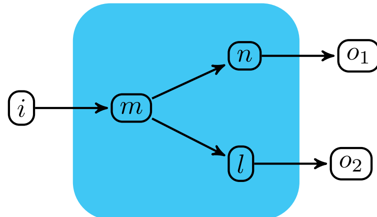
Andrea Cerone[1] and Matthew Hennessy[2]

[1]IMDEA Software Institute, [2]Trinity College Dublin
Andrea.Cerone@imdea.org, Matthew.Hennessy@scss.tcd.ie

Abstract. We present a process calculus for both specifying the desired behaviour of distributed systems and for describing their actual implementation; the calculus is aimed at the internet layer of the TCP/IP reference model. This allows us to define behavioural preorders in the style of DeNicola and Hennessy, relating specifications and implementations for distributed systems at this level of abstraction. The main result of the paper is a complete characterisation of these preorders, for a large class of systems, in terms of traces of extensional actions. This result underpins a sound and complete proof methodology which is demonstrated by the verification of the correct behaviour of a virtual shared memory protocol.

1 Introduction

Different approaches have been made to the problem of verifying the behaviour of distributed systems. This includes model checking [11,5,14] and process calculi [17,9,7,16,12]. In the latter a language, or calculus, is designed for both specifying desired behaviour, for example of a distributed system, and describing the proposed implementation. Verification then consists of formally proving that the two descriptions are *behaviourally equivalent*. The success of this approach is dependent not only on having a robust notion of *behavioural equivalence*, but also a high-level characterisation of the equivalence which can be used for verification purposes.



We define a process calculus for modelling (distributed) systems at a high level of abstraction, roughly at the level of the *Internet Layer* of the *TCP/IP* reference model [19]. We assume non-blocking broadcast communication between independent computational entities, although the introduction of point-to-point communication would not invalidate our results. A typical system may be seen to the left. It consists of a number of nodes or stations at which code

* Supported by SFI project SFI 06 IN.1 1898.

is executed by independent processes, with an *accessibility* relation between the nodes. So in this example only stations n and l are in the range of broadcasts from station m while o_2 is the only station which can pick up messages broadcast from station l. Thus in general communication is *multicast*, in that messages can be received by multiple entities simultaneously; for example messages transmitted from m can be picked up at both n and l simultaneously. However either of the nodes n, l, or indeed both, can choose to ignore messages, or more generally may be in a state where broadcasts cannot currently be received. Thus broadcasts are *non-blocking* in that messages can be transmitted from m regardless of whether or not anybody is currently listening at n or l.

There are two kinds of nodes. The first, the internal nodes, are those at which code is executing, broadcasting and receiving messages; in our example these are m, n and l. Here, and throughout the rest of the paper, we use shadowing to represent these internal nodes. The second are *interface nodes*, such as i, o_1 and o_2, which are not executing any code, and which can be used either to test the internal behaviour of the system by placing test code there, or more generally combining smaller systems to construct larger ones.

More formally, a system $\mathcal{M}$ is a pair of the form $\Gamma \triangleright M$, where Γ is a directed graph describing the *accessibility relation* between nodes, or the system topology, and M is a system term from a process calculus which describes the code running at the internal nodes. In this paper we focus on a sub-calculus of that of [3,2] in which the station code is non-probabilistic. From these papers we also borrow the (non-trivial) partial operator $\mathcal{M} \not\models \mathcal{N}$ for composing systems to form larger ones. For our purposes, we focus on a relevant, large class of *composable* systems, which do not allow connections between interface nodes. It can be shown that all composable systems can be generated by *atomic systems*, which contain only one node, using the operator $\not\models$. As explained in [3,2], the composite system $\mathcal{M} \not\models \mathcal{N}$ may be viewed as $\mathcal{N}$ extending $\mathcal{M}$ by adding new stations and adding code to execute at the interface nodes of $\mathcal{M}$; it may also be viewed as $\mathcal{N}$ *blackbox testing* the system $\mathcal{M}$ by placing probing code at its interface.

This compositional view of systems leads in a natural way to an adaptation of the standard testing based preorders [6] which we denote by $\mathcal{M} \sqsubseteq_{\mathrm{may}} \mathcal{N}$ and $\mathcal{M} \sqsubseteq_{\mathrm{must}} \mathcal{N}$. Intuitively the former means that if the application of any *blackbox* test $\mathcal{T}$, represented by the composite system $\mathcal{M} \not\models \mathcal{T}$, can lead to a success, then so can the application to $\mathcal{N}$; the latter, $\mathcal{M} \sqsubseteq_{\mathrm{must}} \mathcal{N}$, on the other hand is determined by the tested processes $\mathcal{M}$, $\mathcal{N}$ guaranteeing success when probed by any testing system $\mathcal{T}$.

Testing preorders can be used to capture the concept of refinement; a system $\mathcal{M}$ corresponds to an implementation of a more abstract system $\mathcal{N}$, known as the specification, if $\mathcal{M} \sqsubseteq_{\mathrm{may}} \mathcal{N}$ and $\mathcal{M} \sqsubseteq_{\mathrm{must}} \mathcal{N}$. However the usefulness of this notion of refinement depends on our ability to reason about these preorders. This is the topic of this paper.

In standard process calculi, such as CCS and CSP, testing preorders can be characterised by the traces, that is sequences of actions, which processes can perform, together with so-called *acceptances* or *failures*. Here however the situation

is much more complicated. For example messages are broadcast asynchronously; in this respect the approach in [7] is of help. But with distributed systems we also have the problem of whether a multicast, that is the simultaneous broadcast of a message to multiple nodes, can be distinguished from a series of individual broadcasts to each of the nodes in turn. More importantly the ability of tests to interrogate the behaviour of a system depends on the distribution of its nodes, the accessibility between them and the accessibility to these internal nodes from the interface.

In this paper we isolate a set of *extensional actions* for systems; their definition is independent of the distribution of the internal nodes in the system and depends only on their accessibility from the interface nodes. They also take into account the multicasts which systems can perform, represented by the set of interface nodes which can receive the broadcast of a value at any point in time.

Traces of such extensional actions provide a sound proof method for the preorder $\sqsubseteq_{\mathrm{may}}$.

Further, for those systems which do not exhibit non-terminating behaviour when considered in isolation, so-called *strongly-convergent systems*, it is sufficient to record deadlocks into traces to obtain a sound proof method for the preorder $\sqsubseteq_{\mathrm{must}}$. As a further contribution, we show that our proof methods are complete; specifically

(1) extensional traces are complete for $\sqsubseteq_{\mathrm{may}}$ for arbitrary systems
(2) deadlock extensional traces are complete for $\sqsubseteq_{\mathrm{must}}$ when restricted to strongly convergent systems which are in addition *finite spanning*, meaning they can reach a finite number of states by performing a transition.

In this extended abstract we concentrate on outlining the proof of (2) which relies on exhibiting characteristic tests for (deadlock) traces of extensional actions. The definition of such tests is non-trivial; one of the main challenges when defining the characteristic test of a trace is that of coordinating the independent activities of its nodes.

As a sample application of our proof techniques, we prove the correctness of a simple virtual shared memory implementation.

The remainder of the paper is organised as follows. The calculus is presented in §2. In §3 we recall the theory of composition for networks of [3], upon which we define the testing preorders $\sqsubseteq_{\mathrm{may}}, \sqsubseteq_{\mathrm{must}}$. In §4 we explain the extensional semantics, and define both the sets of traces and deadlock traces of systems. These are used to determine our characterisations for both testing preorders in §5, where we put an emphasis on the completeness result for $\sqsubseteq_{\mathrm{must}}$. An application of the resulting proof techniques is given in §6, in which we prove the correctness of a virtual shared memory. We end the paper with a brief comparison with related work, in §7.

The topics treated in this paper have been extracted from [2], to which the reader is referred for detailed proofs of the results and further applications of the proof methods to real world scenarios.

2 A calculus for Distributed Systems

Syntax. A directed connectivity graph $\Gamma = \langle \Gamma_V, \Gamma_E \rangle$ consists of a set of vertices Γ_V, representing the set of nodes of a distributed system, and a relation $\Gamma_E \subseteq (\Gamma_V \times \Gamma_V)$ containing the connections between nodes. If $(m, n) \in \Gamma_E$ then node n can receive messages broadcast by m. Given $\Gamma = \langle \Gamma_V, \Gamma_E \rangle$, we use $\Gamma \vdash m$ for $m \in \Gamma_V$, $\Gamma \vdash m \to n$ for $(m, n) \in \Gamma_E$ and $\Gamma \vdash m \not\to n$ for its negation.

A distributed system is modelled as a tuple $\Gamma \triangleright M$, where Γ is a connectivity graph and M is a system term which associates processes to nodes, generated by the grammar below.

$$M, N ::= \mathbf{0} \mid n[\![P]\!] \mid (M \mid N)$$

A (system) term M is a collection of sub-terms of the form $m[\![P]\!]$, which binds process P to node m; the term $\mathbf{0}$ corresponds to the system term in which no nodes are executing processes. The code for processes will be presented shortly. We will often use the metavariables $\mathcal{M}, \mathcal{N}, \cdots$ when referring to a system $\Gamma \triangleright M$.

Let nodes(M) be the set of the node names appearing in M. We only consider the sublanguage of well-formed systems $\Gamma \triangleright M$ such that nodes$(M) \subseteq \Gamma_V$, and such that each node name occurs at most once in M. These constraints ensure that in systems, nodes running code are part of the system topology, and no node can be bound to multiple processes. It is possible for nodes appearing in Γ not to occur in M. At least intuitively, such nodes represent the external environment of a system and will be used to test its behaviour (§3). The set of the external nodes of a system $\Gamma \triangleright M$, formally defined as $\Gamma_V \setminus \text{nodes}(M)$, is called the interface of the system and denoted by $\mathsf{Intf}(\Gamma \triangleright M)$.

The syntax for processes, given below, is a straightforward instance of a standard process calculus, and their constructs should be self explanatory. Here we assume a (at most countable) set of process definitions of the form $A \Leftarrow P$.

$$P,\ Q ::= \mathbf{0} \mid c!\langle e \rangle\,.P \mid c?(x)\,.P \mid \tau.P \mid \omega \mid P + Q \mid \mathsf{if}\ b\ \mathsf{then}\ P\ \mathsf{else}\ Q \mid A(\tilde{x})$$

For processes we assume some language for Boolean expressions, $b, b', \cdots$ which includes variables, $x, y, \cdots$ and the Boolean values, $\{\mathsf{true}, \mathsf{false}\}$. In the standard manner we assume an interpretation function $[\![\cdot]\!]$ which maps all closed Boolean expressions to some Boolean value. In a similar manner we assume another language of (value) expressions, $e, e', \cdots$, which again may contain variables, and values $v, w, \cdots$ from some *finite* value set; this language also comes equipped with an interpretation function, mapping each closed expression into one of the finite set of values.

We also assume a special clause ω which will be used for testing purposes in §3. Any system which has no occurrence of the special clause is called *proper*.

Intensional Semantics. We define a collection of *Structured Operational Semantics* rules, whose judgements take the form $(\Gamma \triangleright M) \xrightarrow{\mu} (\Gamma \triangleright N)$, to define the behaviour of systems. The action μ can have either the form **(a)** $m.c!v$, node

$$\text{(B-BROAD)}\ \frac{P \xrightarrow{c!v} Q}{\Gamma \rhd n[\![P]\!] \xrightarrow{n.c!v} \Gamma \rhd n[\![Q]\!]} \qquad \text{(B-REC)}\ \frac{P \xrightarrow{c?v} Q \qquad \Gamma \vdash m \to n}{\Gamma \rhd n[\![P]\!] \xrightarrow{m.c?v} \Gamma \rhd n[\![Q]\!]}$$

$$\text{(B-DEAF)}\ \frac{P \not\xrightarrow{c?v}}{\Gamma \rhd n[\![P]\!] \xrightarrow{m.c?v} \Gamma \rhd n[\![P]\!]} \qquad \text{(B-DISC)}\ \frac{\Gamma \vdash m \not\to n}{\Gamma \rhd n[\![P]\!] \xrightarrow{m.c?v} \Gamma \rhd n[\![P]\!]}$$

$$\text{(B-}\mathbf{0}\text{)}\ \frac{}{\Gamma \rhd \mathbf{0} \xrightarrow{m.c?v} \Gamma \rhd \mathbf{0}} \qquad \text{(B-}\tau\text{)}\ \frac{P \xrightarrow{\tau} Q}{\Gamma \rhd n[\![P]\!] \xrightarrow{\tau} \Gamma \rhd n[\![Q]\!]}$$

$$\text{(B-}\tau\text{.PROP}-\text{L)}\ \frac{\Gamma \rhd M \xrightarrow{\tau} \Gamma \rhd L}{\Gamma \rhd M \mid N \xrightarrow{\tau} \Gamma \rhd L \mid N} \qquad \text{(B-}\tau\text{.PROP}-\text{R)}\ \frac{\Gamma \rhd N \xrightarrow{\tau} \Gamma \rhd L}{\Gamma \rhd M \mid N \xrightarrow{\tau} \Gamma \rhd M \mid L}$$

$$\text{(B-SYNC)}\ \frac{\Gamma \rhd M \xrightarrow{\mu_1} \Gamma \rhd M' \quad \Gamma \rhd N \xrightarrow{\mu_2} \Gamma \rhd N'}{\Gamma \rhd M \mid N \xrightarrow{(\mu_1 \circ \mu_2)} \Gamma \rhd M' \mid N'}$$

$\mu_1 \circ \mu_2$	$m.c!v$	$m.c?v$
$m.c!v$		$m.c!v$
$m.c?v$	$m.c!v$	$m.c?v$

Fig. 1. Labelled Transition Semantics for (high level) systems

$$\text{(S-SND)}\ \frac{[\![e]\!] = v}{c!\langle e\rangle\,.P \xrightarrow{c!v} P} \qquad \text{(S-RCV)}\ \frac{}{c?(x)\,.P \xrightarrow{c?v} \{v/x\}P}$$

$$\text{(S-}\tau\text{)}\ \frac{}{\tau.P \xrightarrow{\tau} P} \qquad \text{(S-SUM}-\text{L)}\ \frac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'}$$

$$\text{(S-THEN)}\ \frac{P \xrightarrow{\alpha} P' \qquad [\![b]\!] = \mathsf{true}}{\mathsf{if}\ b\ \mathsf{then}\ P\ \mathsf{else}\ Q \xrightarrow{\alpha} P'} \qquad \text{(S-PDEF)}\ \frac{A(\tilde{x}) \Leftarrow P \qquad \{\tilde{e}/\tilde{x}\}P \xrightarrow{\alpha} Q}{A\langle\tilde{e}\rangle \xrightarrow{\alpha} Q}$$

Fig. 2. Pre-semantics of processes

m broadcasts value v along channel c, **(b)** $m.c?v$, the system receives a broadcast of value v along channel c from the external node m, or **(c)** τ, some node performs an internal activity.

The rules of the labelled transition semantics are depicted in Figure 1. They are based on a pre-semantics for processes, whose rules are given in Figure 2 and should be self-explanatory. Some symmetric rules have been omitted.

Rule (B-BROAD) models a node which is willing to transmit a value v, while the next four rules deal with how stations react to such a broadcast. When a station is within the transmitter's range and is waiting to detect a value along channel c, it will receive it correctly (Rule (B-REC)). On the other hand, if either the station is not waiting to detect a value along channel c (Rule (B-DEAF)) or is not in the sender's transmission range (rule (B-DISC)), then the broadcast is ignored. In rule (B-DEAF), $P \not\xrightarrow{c?v}$ means that $P \xrightarrow{c?v} Q$ for no process Q. Finally, the $\mathbf{0}$ system term ignores all transmissions (Rule (B-$\mathbf{0}$)).

Rule (B-SYNC) models how nodes interact. The action performed by a system $(\Gamma \rhd M \mid N)$ is determined by the individual actions performed by $\Gamma \rhd M$ and

$\Gamma \triangleright N$, according to a (partial) binary operator for actions $\circ$ defined at the right of Rule (B-SYNC) in Figure 1. Here it is important to note that the action induced by a transmission and a reception is again a transmission, thus implementing broadcast communication; see [17] for a detailed discussion. The remaining rules, modelling internal activity, are straightforward.

Example 1. Consider the system $\mathcal{M} = \Gamma \triangleright M$, where Γ is the directed graph depicted on page 1, M is $m[\![A]\!] \,|\, n[\![A]\!] \,|\, l[\![A]\!]$ and $A \Leftarrow c?(x)\,.c!\langle x\rangle$. All the internal nodes m, n, l are waiting to receive a value via channel c in $\mathcal{M}$; once any of these nodes has received such a value, it will forward it along the same channel. One possible behaviour of $\mathcal{M}$ can be summarised as follows; first node m detects a broadcast of an arbitrary value v along channel c, performed by node i. Nodes n, l are not affected by this broadcast, since they are not in the range of transmission of i. Using Rules (B-REC), (B-DISC) and (B-SYNCH) we can infer the transition $\mathcal{M} \xrightarrow{i.c?v} \mathcal{M}_1$, where $\mathcal{M}_1 = \Gamma \triangleright m[\![P_v]\!] \mid n[\![A]\!] \mid l[\![A]\!]$ and $P_v = c!\langle v\rangle$.

Next node m forwards value v to both nodes n and l, which are in its range of transmission. This is formalised by the transition $\mathcal{M}_1 \xrightarrow{m.c!v} \mathcal{M}_2$, where $\mathcal{M}_2 = \Gamma \triangleright m[\![\mathbf{0}]\!] \mid n[\![P_v]\!] \mid l[\![P_v]\!]$, which is obtained using rules (B-BROAD), (B-REC) and (B-SYNCH). Finally, we have a broadcast fired by n followed by one fired by l. Let $\mathcal{M}_3 = \Gamma \triangleright m[\![\mathbf{0}]\!] \mid n[\![\mathbf{0}]\!] \mid l[\![P_v]\!]$ and $\mathcal{M}_4 = \Gamma \triangleright m[\![\mathbf{0}]\!] \mid n[\![\mathbf{0}]\!] \mid l[\![\mathbf{0}]\!]$; then we have $\mathcal{M}_2 \xrightarrow{n.c!v} \mathcal{M}_3 \xrightarrow{l.c!v} \mathcal{M}_4$. □

Reduction Semantics. In the following we will want to discuss the behaviour of systems when isolated from its interface; that is, when input actions of interface nodes are inhibited. To this end, we define a reduction relation $\rightharpoonup$ by letting $\mathcal{M} \rightharpoonup \mathcal{N}$ if either $\mathcal{M} \xrightarrow{m.c!v} \mathcal{N}$ or $\mathcal{M} \xrightarrow{\tau} \mathcal{N}$; any maximal sequence of reductions rooted in a system $\mathcal{M}$ is called a computation for $\mathcal{M}$. Note that reductions $\rightharpoonup$ can be defined directly via a reduction semantics; see [2], §2.2 and §2.4.

3 Testing Distributed Systems

Extension of Systems Following the approach of [3] we focus on a specific class of systems and how they can be extended to obtain larger systems.

Definition 1 (Composable Systems). *A system $\mathcal{M} = \Gamma \triangleright M$ is* composable *if whenever $\Gamma \vdash m \rightarrow n$ then either $m \in nodes(M)$ or $n \in nodes(M)$.* □

Henceforth we will always assume that systems are composable. Such systems do not allow connections between nodes in their interface; from the point of view of a system, the only visible information about its external environment consists of the set of access points to the system. In our terminology its interface.

Focusing on composable systems is not restrictive, in that connections between external nodes do not affect the behaviour of systems, so that any system $\mathcal{M}$ can be reduced to a composable one $\mathcal{M}'$ without affecting the transitions it can perform. Below we define an extension operator which allows us to infer a larger system from a given one, in which nodes which were external in the latter may now be connected.

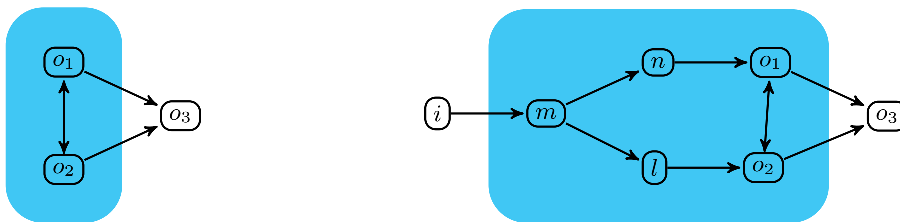


Fig. 3. Two systems $\mathcal{N}$ and $\mathcal{L} = (\mathcal{M} \not\triangleright \mathcal{N})$.

Definition 2 (Systems Extension). *Let $\mathcal{M} = (\Gamma_M \triangleright M), \mathcal{N} = (\Gamma_N \triangleright N)$ be (composable) systems; the extension of $\mathcal{M}$ with $\mathcal{N}$, $(\mathcal{M} \not\triangleright \mathcal{N})$, is defined whenever $\mathsf{nodes}(M) \cap (\Gamma_N)_V = \emptyset$, and it is equivalent to $(\Gamma_M \cup \Gamma_N) \triangleright (M \mid N)$, where $\Gamma_M \cup \Gamma_N$ is defined as the pointwise union of their sets of vertices and edges.* □

Intuitively, $\mathcal{M} \not\triangleright \mathcal{N}$ describes an extension of the system $\mathcal{M}$, where the information about its external environment is supplied by a second system $\mathcal{N}$; such system can contain the code run by interface nodes of $\mathcal{M}$, and the connections between its interface nodes. But it can also contain new nodes, which did not appear in $\mathcal{M}$. Also, in the composite system, the topological structure of $\mathcal{M}$ is left unchanged; this property is desirable, as we wish to use the operator $\not\triangleright$ to implement blackbox testing. In [3] we proved that $\not\triangleright$ is associative and closed with respect to the set of composable systems, and that it is the most expressive operator which can be used to implement blackbox testing.

Example 2. Let $\mathcal{M}$ be the system of Example 1; its interface $\mathsf{Intf}(\mathcal{M}) = \{i, o_1, o_2\}$ represents its external environment. By using the extension operator $\not\triangleright$ we can obtain a new system which contains $\mathcal{M}$, and which also gives new information about the connections of the nodes in the external environment of $\mathcal{M}$.

For example, consider the system $\mathcal{N} = \Gamma_N \triangleright o_1[\![A]\!] \mid o_2[\![A]\!]$, depicted on the left of Figure 3; here A is as defined in Example 1. This system specifies the code nodes o_1, o_2 are running, together with their connections; it also has a fresh node o_3 in its own interface. The composite system $\mathcal{L} = (\mathcal{M} \not\triangleright \mathcal{N})$ is well-defined, and it models the system obtained by extending $\mathcal{M}$ with the information regarding its external environment provided by $\mathcal{N}$. The system $\mathcal{L}$ is depicted to the right of Figure 3. Here the sub-system $\mathcal{N}$ could be viewed as a black box tester for the original system $\mathcal{M}$, placing probing code at two of $\mathcal{M}$'s interface nodes, and having another node o_3 where the results of this probing could be collected. □

Testing Preorders. We say that a system $\mathcal{M}$ is successful if the clause ω appears unguarded in the code of one of its nodes, while a computation of a system $\mathcal{M}$ is said to be successful if it contains a successful system. Given two systems $\mathcal{M}, \mathcal{T}$ we say that $\mathcal{M}$ **may-pass** $\mathcal{T}$ if $\mathcal{M} \not\triangleright \mathcal{T}$ has a successful computation, while $\mathcal{M}$ **must-pass** $\mathcal{T}$ if all its computations are successful.

Definition 3 (Testing Preorders). *Let $\mathcal{M},\mathcal{N}$ be two systems; we say that $\mathcal{M} \sqsubseteq_{may} \mathcal{N}$ if for any $\mathcal{T}$ which can be used to extend both $\mathcal{M},\mathcal{N}$ we have that $\mathcal{M}$ **may-pass** $\mathcal{T}$ implies $\mathcal{N}$ **may-pass** $\mathcal{T}$. Similarly, we define $\mathcal{M} \sqsubseteq_{must} \mathcal{N}$ in the same way, this time using the **must-pass** testing relation. We say that $\mathcal{M} \sqsubseteq \mathcal{N}$ if both $\mathcal{M} \sqsubseteq_{may} \mathcal{N}$ and $\mathcal{M} \sqsubseteq_{must} \mathcal{N}$ are true; finally, we use the notation $\mathcal{M} \simeq \mathcal{N}$ if both $\mathcal{M} \sqsubseteq \mathcal{N}$ and $\mathcal{N} \sqsubseteq \mathcal{M}$ are true.* □

Example 3 (Deadlocks). Let Γ represent the system topology containing two nodes m, e and having as its only connections $\Gamma \vdash m \to e, e \to m$. Let $\mathcal{M} = \Gamma \triangleright m[\![P]\!], \mathcal{N} = \Gamma \triangleright m[\![Q]\!]$, where $P = c!\langle v\rangle$, $Q = c?(x)\,.c!\langle v\rangle$. Both systems have the ability to broadcast value v along channel c; however, in $\mathcal{N}$ this broadcast is enabled only after a broadcast along channel c has been performed by node e. That is, $\mathcal{N} \twoheadrightarrow \mathcal{N}'$ for no $\mathcal{N}'$.

These two systems can be distinguished via the **must-pass** testing relation by the test $\mathcal{T} = \Gamma_e \triangleright e[\![c?(x)\,.\omega]\!]$; here Γ_e is the graph with the single node e. Note that $\mathcal{M}$ **must-pass** $\mathcal{T}$, while $\mathcal{N}$ **must-pass** $\mathcal{T}$ is not true. Therefore $\mathcal{M} \not\sqsubseteq_{\text{must}} \mathcal{N}$.

It is also possible to exhibit a test which $\mathcal{N}$ **must-pass**es, but $\mathcal{M}$ does not. To this end, let $\mathcal{T}' = \Gamma_e \triangleright e[\![c?(x)\,.\,\mathbf{0} + \tau.\omega]\!]$. It is easy to see that $\mathcal{M} \not\triangleright \mathcal{T}'$ has an unsuccessful computation; however, since $\mathcal{N}$ is deadlocked and cannot broadcast a value, the only possibility for $\mathcal{N} \not\triangleright \mathcal{T}'$ is that the testing component performs a τ action, thus entering a successful state. $\mathcal{N}$ **must-pass** $\mathcal{T}'$, hence $\mathcal{N} \not\sqsubseteq_{\text{must}} \mathcal{M}$.

What distinguishes $\mathcal{M}$ from $\mathcal{N}$ is that the latter cannot broadcast value v without first receiving a value from node e first, that is it is deadlocked. However suppose that we add the possibility to $\mathcal{N}$ to directly broadcast value v, leading to the system $\mathcal{N}' = \Gamma \triangleright m[\![Q']\!]$, where $Q' = Q + c!\langle v\rangle$; as we will see, it is impossible to distinguish $\mathcal{N}'$ from $\mathcal{M}$. Note that $\mathcal{N}'$ is not deadlocked.

Also, note that P, Q' are valid value-passing CCS processes, and that their sets of acceptances are different. In fact, in value-passing CCS, P **must-pass**es the test $c!\langle v\rangle + \tau.\omega$, but the same is not true for Q'; also, Q' **must-pass**es the test $c!\langle v\rangle\,.\omega$, but this is not true for P. This example gives an intuition that deadlocks, rather than acceptances, should be taken into account when giving a characterisation of the preorder $\sqsubseteq_{\text{must}}$. □

4 Extensional Semantics

The extensional semantics of systems is defined in Figure 4; its transitions $\mathcal{M} \overset{\lambda}{\longmapsto} \mathcal{N}$ can be either **(a)** internal activities τ, **(b)** inputs $i.c?v$ performed by an input node i or **(c)** broadcast actions $c!v \triangleright \eta$, observable at a non-empty set of output nodes η. Note that any node name mentioned in the action of the transition occurs in the interface of the source configuration $\mathcal{M}$. The set of extensional actions is denoted by EAct_τ, while $\text{EAct} = \text{EAct}_\tau \setminus \{\tau\}$.

Rules (s-Tau) and (s-Shh) model unobservable activities. The first rule propagates internal activities of nodes to systems, while the second rule states that broadcasts which cannot be detected by any external node of a system cannot be observed. Rule (s-In) propagates input actions to the extensional semantics;

$$(\text{s-Tau})\ \frac{\mathcal{M} \xrightarrow{m.\tau} \mathcal{N}}{\mathcal{M} \overset{\tau}{\longmapsto} \mathcal{N}} \qquad (\text{s-Shh})\ \frac{\mathcal{M} \xrightarrow{m.c!v} \mathcal{N} \qquad \{n \in \mathsf{Intf}(\mathcal{M}) \mid \mathcal{M} \vdash m \to n\} = \emptyset}{\mathcal{M} \overset{\tau}{\longmapsto} \mathcal{N}}$$

$$(\text{s-In})\ \frac{\mathcal{M} \xrightarrow{i.c?v} \mathcal{N}}{\mathcal{M} \overset{i.c?v}{\longmapsto} \mathcal{N}} \qquad (\text{s-Out})\ \frac{\mathcal{M} \xrightarrow{m.c!v} \mathcal{N} \qquad \eta := \{n \in \mathsf{Intf}(\mathcal{M}) \mid \mathcal{M} \vdash m \to n\} \neq \emptyset}{\mathcal{M} \overset{c!v \triangleright \eta}{\longmapsto} \mathcal{N}}$$

Fig. 4. Extensional Semantics

finally, Rule (s-Out) models outputs which can be observed by a set of external nodes η.

The extensional semantics endows systems with the structure of a LTS [13], which we call the extensional LTS of systems. Most of the terminology used for LTSs in the literature can be then readapted to systems. However our definition of *weak* extensional actions, taken from [3] needs to be non-standard.

Definition 4 (Weak Extensional Actions). *For any systems $\mathcal{M}, \mathcal{N}$, we say that $\mathcal{M} \overset{\tau}{\Longmapsto} \mathcal{N}$ if $\mathcal{M} \overset{\tau}{\longmapsto}^{*} \mathcal{N}$; here $\overset{\tau}{\longmapsto}^{*}$ is the reflexive, transitive closure of $\overset{\tau}{\longmapsto}$. Further, we say that $\mathcal{M} \overset{i.c?v}{\Longmapsto} \mathcal{N}$ if $\mathcal{M} \overset{\tau}{\Longmapsto} \overset{i.c?v}{\longmapsto} \overset{\tau}{\Longmapsto} \mathcal{N}$. Finally, we say that $\mathcal{M} \overset{c!v \triangleright \eta}{\Longmapsto} \mathcal{N}$ if either $\mathcal{M} \overset{\tau}{\Longmapsto} \overset{c!v \triangleright \eta}{\longmapsto} \overset{\tau}{\Longmapsto} \mathcal{N}$, or if there exist two non-empty sets of nodes η_1, η_2 such that $\eta_1 \cup \eta_2 = \eta, \eta_1 \cap \eta_2 = \emptyset$ and $\mathcal{M} \overset{c!v \triangleright \eta_1}{\Longmapsto} \overset{c!v \triangleright \eta_2}{\Longmapsto} \mathcal{N}$.*

These single weak transitions are extended to sequences $\mathcal{M} \overset{s}{\Longmapsto} \mathcal{N}$, for $s \in EAct^{*}$, in the obvious manner. □

The complication in the definition of $\mathcal{M} \overset{c!v \triangleright \eta}{\Longmapsto} \mathcal{N}$ is necessary in order to be able to simulate a multicast $c!v \triangleright \eta$, where the set η contains more than one node name, by a sequence of single broadcasts; this is shown in the Example below.

Example 4. Let $\mathcal{M} = \Gamma_M \triangleright m[\![c!v]\!]$, where $\Gamma_M \vdash m \to o_1, m \to o_2$. Also, let $\mathcal{N} = \Gamma_N \triangleright m[\![c!v]\!] \mid n[\![c!v]\!]$, where $\Gamma_N \vdash m \to o_1, n \to o_2$. Both systems are able to deliver value v, along channel c, to the interface nodes o_1, o_2. However, while $\mathcal{M}$ does it with a single broadcast, $\mathcal{N}$ uses two broadcasts which can be detected by nodes o_1, o_2 individually.

One could expect that there is a test that allows us to distinguish $\mathcal{M}$ from $\mathcal{N}$, in the sense of $\sqsubseteq_{\text{may}}$. However, at least intuitively, the broadcast of $\mathcal{M}$ can be simulated in system $\mathcal{N}$, by firing the two broadcasts in sequence. That is, the action $\mathcal{M} \overset{c!v \triangleright \{o_1,o_2\}}{\Longmapsto}$ can be matched by another action $\mathcal{N} \overset{c!v \triangleright \{o_1,o_2\}}{\Longmapsto}$, where the latter can be inferred from $\mathcal{N} \overset{c!v \triangleright \{o_1\}}{\Longmapsto} \overset{c!v \triangleright \{o_2\}}{\Longmapsto}$ using our non-standard definition of weak extensional actions.

If a test $\mathcal{T}$ is used to test the system $\mathcal{M}$, each configuration $\mathcal{T}'$ reached by the test $\mathcal{T}$ after $\mathcal{M}$ has broadcast value v, can be also obtained when $\mathcal{T}$ is used

to test the system $\mathcal{N}$, by letting the latter fire its two broadcasts in sequence. That is, $\mathcal{M} \sqsubseteq_{\text{may}} \mathcal{N}$.

On the other hand, $\mathcal{N} \not\sqsubseteq_{\text{may}} \mathcal{M}$. To prove this, it is sufficient to provide a test $\mathcal{T}$ which $\mathcal{N}$ **may-pass**es, but $\mathcal{M}$ does not. The reader can easily check that $\Gamma_T \triangleright o_1[\![c?(x)\,.c!\langle w\rangle]\!] \mid o_2[\![c?(x)\,.c?(y)\,.\mathsf{if}\ (x = y)\ \mathsf{then}\ \mathbf{0}\ \mathsf{else}\ \omega]\!]$, where $\Gamma_T \vdash o_1 \to o_2$, is one such test.

A similar argument shows that $\mathcal{N} \sqsubseteq_{\text{must}} \mathcal{M}$, while $\mathcal{M} \not\sqsubseteq_{\text{must}} \mathcal{N}$. □

5 Characterisation of the Testing preorders

Traces and Deadlock Traces. A system $\mathcal{M}$ is *finite spanning* if any $\mathcal{N}$ in the extensional LTS generated by $\mathcal{M}$ can reach a finite number of systems by performing a transition. It is *convergent* if it has no infinite computation, *strongly convergent* if every state in the extensional LTS it generates is convergent.

From Example 3 we know that must-testing preorder is sensitive to deadlocks; this is the essential ingredient to the following definition:

Definition 5 (Traces, Deadlock Traces). *Let $\delta \notin EAct$. For any $\mathcal{M}$ we let*

$$Traces(\mathcal{M}) = \{\, s \in EAct^* \mid \mathcal{M} \models\!\!\stackrel{s}{\Longrightarrow} \mathcal{N} \text{ for some } \mathcal{N} \,\}$$

$$Dtraces(\mathcal{M}) = \{\, s{::}\delta \mid \mathcal{M} \models\!\!\stackrel{s}{\Longrightarrow} \mathcal{N} \text{ for some } \mathcal{N}, s \in EAct^* \text{such that } \mathcal{N} \not\triangleright \,\}$$

We used the symbol :: to separate occurrences of elements in lists. □

Example 5. Consider again the system $\mathcal{M},\mathcal{N}$ of Example 3. We have already noted that $\mathcal{M} \not\sqsubseteq_{\text{must}} \mathcal{N}$, and $\mathcal{N} \not\sqsubseteq_{\text{must}} \mathcal{M}$. Also, we have that $\delta \in \text{Dtraces}(\mathcal{N})$ (note that $\mathcal{N} \not\triangleright$), while $\delta \notin \text{Dtraces}(\mathcal{M})$. Further, we have that $c!v \triangleright \{e\}{::}\delta \in \text{Dtraces}(\mathcal{M})$, but this trace is not in $\text{Dtraces}(\mathcal{N})$; in fact, in the latter system the broadcast of value v can happen only after a value has been received along channel c. That is, $c?w{::}c!v \triangleright \{e\}{::}\delta \in \text{Dtraces}(\mathcal{M})$ for an arbitrary value w.

Now consider the system $\mathcal{N}'$ of Example 3. Note that $c!v\triangleright\{e\}{::}\delta \in \text{Dtraces}(\mathcal{N}')$, and $\delta \notin \text{Dtraces}(\mathcal{N}')$. Even more, the two systems $\mathcal{M}$, $\mathcal{N}'$ share the same set of deadlock traces, the minimal fragment of which we are interested in being $\{c!\langle v\rangle{::}\delta, c?(w){::}\delta \mid w \text{ arbitrary value }\}$. Theorem 1, coming up, implies that $\mathcal{M}$ and $\mathcal{N}'$ cannot be distinguished by the **must-pass** relation. □

Full Abstraction. We can now state the main result of the paper:

Theorem 1 (Characterisation of the testing preorders). *Let $\mathcal{M},\mathcal{N}$, be two proper systems. Then*

(1) $\mathcal{M} \sqsubseteq_{may} \mathcal{N}$ if and only if $Traces(\mathcal{M}) \subseteq Traces(\mathcal{N})$,
(2) if $\mathcal{M},\mathcal{N}$ are finite spanning and strongly convergent then $\mathcal{M} \sqsubseteq_{must} \mathcal{N}$ if and only if $Dtraces(\mathcal{M}) \supseteq Dtraces(\mathcal{N})$.

The restriction to proper systems is natural, as the special action ω should only appear in systems used for testing. The restriction to strongly convergent systems in part (2) is needed because in our definition of deadlock traces we did not take divergence into account. Doing so would be quite complicated as, due to the non-blocking nature of broadcasts, it is possible to define two divergent systems $\mathcal{M}, \mathcal{N}$ which are not must-testing related. This is in contrast with the standard theory of must-testing [6]. See [2], Remark 4.4.3, Page 88, for a detailed discussion. See also §4.4.3, Page 98, for a potential solution to this problem.

In this extended abstract we only have space to give an outline of Theorem 1(2). This is split into two parts, soundness and completeness.

Theorem 2 (Soundness for $\sqsubseteq_{\mathbf{must}}$). *If Dtraces($\mathcal{M}$) $\supseteq$ Dtraces($\mathcal{N}$) then $\mathcal{M} \sqsubseteq_{must} \mathcal{N}$, for proper, finite spanning, strongly convergent systems.*

Proof. It suffices to show that inclusion of deadlock traces is preserved when extending systems by adding a new node. That is, if Dtraces($\mathcal{M}$) $\supseteq$ Dtraces($\mathcal{N}$), and $\mathcal{T} = \Gamma_T \triangleright e[\![P]\!]$ is a system such that $\mathcal{M} \not\models \mathcal{T}$ and $\mathcal{N} \not\models \mathcal{T}$ are defined, then Dtraces($\mathcal{M} \not\models \mathcal{T}$) $\supseteq$ Dtraces($\mathcal{N} \not\models \mathcal{T}$). See [2], §4.3.1 and §4.4.1, for a detailed proof. □

The standard approach to prove the converse to Theorem 2 is that of providing characteristic tests for deadlock traces.

Proposition 1 (Characteristic Tests). *Let $\mathcal{M}$ be a proper, finite spanning, strongly convergent system. Then for any set of nodes η such that* $\mathsf{Intf}(\mathcal{M}) \subseteq \eta$ *and trace t there exists a system $\mathcal{T}_t^\eta$ such that $t \notin$ Dtraces($\mathcal{M}$) if and only if $\mathcal{M}$* **must-pass** *$\mathcal{T}_t^\eta$. Such a system is called a* characteristic tests *for t, with respect to η.*

Proof. See §4.3.2 and §4.4.2 of [2]. □

Characteristic tests are parameterised by a set of nodes η; in general the topology of a test depends on the interface of the system being tested. The formal definition of the characteristic tests $\mathcal{T}_t^\eta$ is given in Appendix A, while here we focus on an informal explanation of their behaviour.

A characteristic test $\mathcal{T}_t^\eta$ contains all the nodes in η and a fresh node cn, called controller node. The controller node is connected (in both directions) with each of the nodes in η. The test $\mathcal{T}_t^\eta$ tests whether a system does not exhibit the deadlock trace t by testing sequentially for every action included in t (or for deadlock when testing for δ), then declaring success whenever it determines that the current action being tested cannot be performed by the tested system.

To achieve this, in $\mathcal{T}_t^\eta$ the nodes in η constantly report (via a fresh channel) the observed behaviour of the tested system to the controller node; these partial information are then used by the latter to infer whether the (extensional) action being tested has been performed by the tested system, in which case $\mathcal{T}_t^\eta$ proceeds by testing for the next action in t. In the process of inferring whether an extensional action has been performed by the tested system, the controller

node also asks nodes in η to report the absence of observed behaviour. For example, detecting an action of the form $c!v \triangleright \eta'$ requires that no node in $\eta \setminus \eta'$ observed a broadcast. Or, when testing for deadlock, no node in η should detect a broadcast. Below we give a detailed explanation of the protocol used by $\mathcal{T}_t^\eta$ to detect an extensional action, with a particular emphasis to an output action of the form $c!v \triangleright \eta'$; the protocol runs in three stages.

Detect: The controller node waits for a relevant subset of nodes in η, consisting of those nodes mentioned in the action being tested, to report to the controller node that their local contribution to the action being tested has been performed. In the case of an action of the form $c!v \triangleright \eta'$, node cn awaits a message from each of the nodes in cn. Each of such nodes broadcasts to the cn only if it detects a broadcast of value v along channel c.

Check: The controller node requests to all the nodes in η whether they observed any other additional activity from the tested system, and awaits a response from each of such nodes; in contrast, the latter do not answer to such a request if they observed some activity from the tested system. In the case of an action of the form $c!v \triangleright \eta'$, nodes in $\eta' \setminus \eta$ do not answer whether they detected a broadcast performed by the tested system.

Proceed: The controller node sends a request to all nodes in η that it is ready to detect the next action in the trace being tested. If nodes in η observe some activity performed by the testee, they deadlock, causing the execution of $\mathcal{T}^\eta$ to be eventually unable to proceed.

Failure: At any given point, the controller node can non-deterministically declare success, exception made for when it detects that the entire trace being tested has been performed by the testee. In this case $\mathcal{T}_t^\eta$ deadlocks, thus causing the test to fail.

Theorem 3 (Completeness for $\sqsubseteq_{\mathbf{must}}$). *$\mathcal{M} \sqsubseteq_{must} \mathcal{N}$ implies Dtraces$(\mathcal{M}) \supseteq$ Dtraces$(\mathcal{N})$, assuming both are finite spanning, strongly convergent, proper systems.*

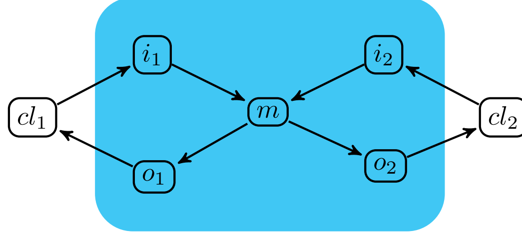
Proof. Suppose $\mathcal{M} \sqsubseteq_{\mathrm{must}} \mathcal{N}$ is true and $t \in \mathrm{Dtraces}(\mathcal{N})$; we need to show that $t \in \mathrm{Dtraces}(\mathcal{M})$. This will follow from the previous proposition if $\mathcal{M}$ **must-pass** $\mathcal{T}_t^\eta$ is not true, where we choose η to be $\mathsf{Intf}(\mathcal{M}) \cup \mathsf{Intf}(\mathcal{N})$.

But this follows form the assumption $\mathcal{M} \sqsubseteq_{\mathrm{must}} \mathcal{N}$ because again the previous proposition gives that $\mathcal{N}$ **must-pass** $\mathcal{T}_t^\eta$ is false. □

6 Application: Virtual Shared Memory

To show the usefulness of our results and proof methodologies, we prove the correctness of a *Virtual Shared Memory* (**VSM**) protocol without replicas. To keep the discussion simple, we consider the case in which the VSM is accessed by two users; however, our case study can be generalised to an arbitrary number of users. We only give an informal description of the specification and the implementation we provide for VSMs, while we defer all formal definitions to Appendix B.

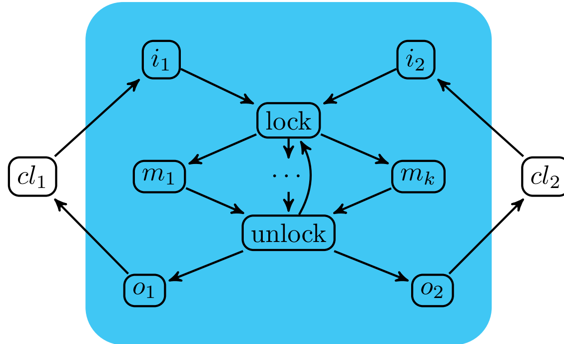
The Specification. In a virtual shared memory protocol, a distributed system provides to two or more users the ability to write and read memory locations which are physically stored at different nodes, while giving them the illusion that the whole memory is stored at a single node.



The system $\mathcal{V}$ specifying a VSM is depicted on the left. Nodes i_j, o_j, where $j = 1, 2$, regulate the flow of messages between a client cl_j and the site where the memory is actually stored, m. Node i_j discards any ill-formed request issued by client cl_j, while o_j ensures that the answer by m corresponds to a request originally performed by cl_j. More specifically, let $j = 1, 2$. Upon receiving a message of the form **read**(x), or **write**(x, n), node i_j will store it in a local queue. At any given point, it will dequeue the next element stored and will forward it through channel c_j. Node m is equipped with two queues, q_1 and q_2; upon receiving a request from node i_j, it will enqueue it in q_j. At any given point, node m can dequeue the next request from one of the queues q_k, $k = 1, 2$. If such a request has the form read x, then it will forward the value stored at its memory location x along channel d_k. If the request has the form write(x,n), node m will update the content of variable x to n, then it will broadcast the new value of x along channel d_k. Upon receiving a message along channel d_k, node o_k stores it in a local queue. At any given point, it non-deterministically dequeues the next stored element, and forwards it to client cl_k.

Since our proof methods are sound only for finite spanning systems, we can assume that the size of the memory stored at node m is finite, since the nodes cl_j, $j = 1, 2$ can only send a finite amount of requests to the system $\mathcal{V}$.

The Implementation. Next we turn our attention to a possible distributed implementation of our VSM. The idea is that of partitioning the memory stored at node m of the specification, among different nodes, $m_1, \cdots, m_k$.



Each node contains only a subset of the locations stored in the total memory, and no memory location is stored in more than one node. To ensure that requests are processed in the same order in which they have been issued by a client, the access to nodes $m_1, \cdots, m_k$ is regulated by two nodes, called *lock* and *unlock*. These two nodes ensure that nodes $m_1, \cdots, m_k$ never handle two requests concurrently.

In the implementation $\mathcal{I}$, the behaviour of nodes cl_j, i_j and o_j, where $j = 1, 2$, is the same as in the specification $\mathcal{V}$. Therefore we concentrate on describing the behaviour of the other nodes. Node lock is equipped with two queues q_1, q_2. Upon receiving a request along channel c_j, $j = 1, 2$, it will store it in the queue q_j. Node lock is also equipped with a boolean flag l, initially set to true. At any given time, node lock checks if l is set to true; in this case, it sets the flag to false, then it selects a queue q_k, $k = 1, 2$, dequeues the next message stored in it, and broadcasts it along channel c_k. Node lock will then wait to receive a message along channel d_k, before resetting the flag l to false. Any message broadcast by node lock along channels d_k will be received by all nodes $m_1, \cdots, m_k$. Such a message contains either a request to read and write some memory location x, which is stored in exactly one of such nodes, say m_n. Node m_n will reply to this request, updating the contents of location x, if needed, by broadcasting a message along channel d_k, which will in turn received by node unlock. Upon receiving a message along channel d_k, node unlock immediately broadcasts it along the same channel, causing both the message to be delivered at node o_k, and node lock to set its internal flag to true.

In Appendix B we give a formal description in our language of both the Specification, as the system $\mathcal{V}$, and the distributed Implementation, as the system term $\mathcal{I}$. The proof that the Implementation satisfies the Specification consists in establishing that $\mathcal{I} \simeq \mathcal{V}$. This is achieved by using the alternative characterisations of the preorders in terms of traces and Deadlock traces. A more detailed explanation of both the specification and implementation of our virtual shared memory can be found in [2], §5.5 at Page 124.

Theorem 4. *Dtraces*$(\mathcal{I})$ = *Dtraces*$(\mathcal{V})$ *and Traces*$(\mathcal{I})$ = *Traces*$(\mathcal{V})$. *In particular,* $\mathcal{I} \simeq \mathcal{V}$. □

7 Conclusions

The achievements of this paper have been more theoretical than practical, although we have also provided some evidence of applicability, via a case study. Many other applications can be found in Chapter 5 of [2]; these include connectionless and connection-oriented routing, for which implementations at different levels of the TCP/IP reference model have been provided, and multicast routing.

To the best of our knowledge, this paper presents the first completeness result for testing preorders applied to distributed systems. The proof is non-trivial, requiring the isolation of our *extensional actions*, and the detailed programming of the characteristic testing contexts. These probe systems via the *extensional actions* and then combine the results of these probes to elicit behavioural characteristics. The ability to define these characteristic tests also depends on the level of our abstraction of our system descriptions. It is far from clear if the completeness result remains true at other levels of abstraction, or if more general systems are considered. For example we already know from [3] that if we add probabilistic behaviour to nodes then the natural generalisation to probabilistic simulations no longer characterises the (probabilistic) testing preorders.

Indeed it was this surprising phenomenon which prompted the research reported in the current paper. So it will be interesting to see what extra constructs can be added to our current non-probabilistic calculus without invalidating our characterisation results. Possibilities include introducing features which can be found in wireless networks, such as node mobility [18,15,8], or introducing time and/or collisions, as in [12,4,10,20,1].

References

1. M. Bugliesi, L. Gallina, A. Marin, S. Rossi, and S. Hamadou. Interference-sensitive preorders for manets. In *QEST*, pages 189–198, 2012.
2. A. Cerone. *Foundations of Ad Hoc Wireless Networks*. PhD thesis, Trinity College Dublin, 2012. `http://software.imdea.org/~andrea.cerone/works/thesis.pdf`.
3. A. Cerone and M. Hennessy. Modelling probabilistic wireless networks. *LMCS*, 9(3), 2013.
4. Andrea Cerone, Matthew Hennessy, and Massimo Merro. Modelling mac-layer communications in wireless systems (extended abstract). In *COORDINATION*, pages 16–30, 2013.
5. Conrado Daws, Marta Z. Kwiatkowska, and Gethin Norman. Automatic verification of the ieee-1394 root contention protocol with kronos and prism. *Electr. Notes Theor. Comput. Sci.*, 66(2):104–119, 2002.
6. R. De Nicola and M. Hennessy. Testing equivalences for processes. *TCS*, 34:83–133, 1984.
7. C.F. Ene and T. Muntean. Testing theories for broadcasting processes. *Sci. Ann. Cuza Univ*, 11:214–230, 2002.
8. F. Ghassemi, W. Fokkink, and A. Movaghar. Equational reasoning on mobile ad hoc networks. *Fund. Inf.*, 105(4):375–415, 2010.
9. M. Hennessy and J. Rathke. Bisimulations for a calculus of broadcasting systems. *TCS*, 200, 1998.
10. I. Lanese and D. Sangiorgi. An operational semantics for a calculus for wireless systems. *TCS*, 411(19):1928–1948, 2010.
11. Flavio Lerda and Riccardo Sisto. Distributed-memory model checking with spin. In *SPIN*, pages 22–39, 1999.
12. M. Merro, F. Ballardin, and E. Sibilio. A timed calculus for wireless systems. *TCS*, 412(47):6585–6611, 2011.
13. R. Milner. A calculus of communicating systems. *LNCS*, 92, 1980.
14. Madanlal Musuvathi and Dawson R. Engler. Model checking large network protocol implementations. In *NSDI*, pages 155–168, 2004.
15. S. Nanz and C. Hankin. A framework for security analysis of mobile wireless networks. *TCS*, 367, 2006.
16. Sebastian Nanz and Chris Hankin. Static analysis of routing protocols for ad-hoc networks, March 25 2004.
17. K. Prasad. A calculus of broadcasting systems. *SCP*, 25, 1995.
18. A. Singh, C.R. Ramakrishnan, and S.A. Smolka. A process calculus for mobile ad hoc networks. *SCP*, 75(6):440–469, 2010.
19. Andrew S Tanenbaum. *Computer Networks. 4th Ed.* Prentice Hall PTR, 2002.
20. M. Wang and Y. Lu. A timed calculus for mobile ad hoc networks. *arXiv preprint arXiv:1301.0045*, 2013.

A Definition of the Characteristic Tests

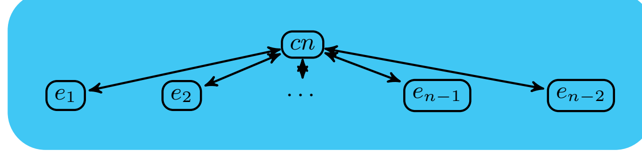


Fig. 5. Topological structure of characteristic tests.

Any characteristic tests $\mathcal{T}_t^\eta$ for deadlock traces takes the form $\Gamma^\eta \triangleright T_t^\eta$; here $\Gamma_v^\eta = \eta \cup \{cn\}$, where cn is a fresh node name and $\Gamma^e ta_v \vdash \{e_1, \cdots, e_k\}$. A graphical representation of Γ_η is given in Figure 5.

The system term T_t^η is defined as $\left(\prod_{i=1}^{k} e_i[\![P_i^t]\!]\right) | cn[\![Q_t]\!]$, where the processes P_i^t and Q^t are defined inductively on the trace t. Before giving the definitions of the processes P_i^t and Q^i, we define some processes which will help maintaining the notation easy. Let $\mathbb{P} = \{P_1, \cdots, P_2, \cdots\}$ be a set of processes. For any $k = 1, 2, \cdots$ we let

$$\mathsf{INFSUM}_{\mathbb{P}}^i \Leftarrow P_i + \mathsf{INFSUM}_{\mathbb{P}}^{i+1}$$

With an abuse of notation, we define $\sum_{i=1}^{\infty} P_i = \mathsf{INFSUM}_{\mathbb{P}}^1$.

Next we give the following process definitions, assuming that the set of channels is $\{c_1, c_2, \cdots\}$.

$$\mathsf{LOCK} \Leftarrow \sum_{i=1}^{\infty} c_i?(x)\,.\,\mathbf{0}$$

$$\mathsf{ALLOW}(c_k, x)\,P \Leftarrow \left(\sum_{\substack{i=1\\ i\neq k}}^{\infty} c_i?(x)\,.\,\mathbf{0}\right) + c_k?(x)\,.P$$

Finally, we adopt the following macros for processes, which encode the housekeeping activities needed by the coordination protocol. First, for any finite list of variables $x_1, \cdots, x_k$ we define the process $c?(x_1, \cdots, x_k)_\omega\,.P$ to be exactly P in the case the list $x_1, \cdots, x_k$ is empty, and $c?(x_1, \cdots, x_k)_\omega\,.P = c?(x_1)\,.(c?(x_2, \cdots, x_k)\,.P) + \tau.\omega$ otherwise. Then, let cc be a fresh channel; the coordination protocol will use the processes defined below; the first three processes will be used by nodes in η, while the remaining

three will be used by the controller node.

$$\begin{aligned}\mathsf{DETECT} &= (cc!\langle \mathsf{ack}\rangle.P) + \mathsf{LOCK}\\ \mathsf{CHECK}.Q &= cc?(x).(cc!\langle \mathsf{clear}\rangle.Q + \mathsf{LOCK}) + \mathsf{LOCK}\\ \mathsf{PROCEED}.Q &= cc?(x).Q + \mathsf{LOCK}\end{aligned}$$

$$\begin{aligned}\mathsf{cnDETECT}(k).Q &= cc_\omega?(x_1,\cdots,x_k).Q\\ \mathsf{cnCHECK}.Q &= cc!\langle \mathsf{check}\rangle.cc?(x_1,\cdots,x_k)_\omega.Q\\ \mathsf{cnPROCEED}.Q &= cc!\langle \mathsf{proceed}\rangle.Q\end{aligned}$$

We are now ready to define the processes P_t^i, Q_t for any trace t. As we have already said, these are defined inductively on the structure of the trace t.

(i) $t = \varepsilon$

$$\begin{aligned}P_\varepsilon^i &\Leftarrow \mathbf{0}\\ Q_\varepsilon &\Leftarrow \mathbf{0}\end{aligned}$$

(ii) $t = \delta$

$$\begin{aligned}P_\delta^i &= \textstyle\sum_{i=1}^{\infty} c_i?(x).\mathsf{DETECT}\\ Q_\delta &= \mathsf{cnDETECT}(1).\,\mathbf{0}\end{aligned}$$

(iii) $t = d!v \triangleright \eta{::}t'$;

$$P_t^i = \begin{cases}\mathsf{ALLOW}(d,x)\ \mathsf{if}\ x \neq v\ \mathsf{then}\ \mathbf{0}\ \mathsf{else} & \\ \quad \mathsf{DETECT.CHECK.PROCEED}.Q_{t'}^i) & \text{if } e_i \in \eta\\ & \\ \mathsf{CHECK.PROCEED}.P_{t'}^i & \text{if } e_j \notin \eta_i\end{cases}$$

$$Q_t = \mathsf{cnDETECT}(|\eta|).\mathsf{cnCHECK}.\mathsf{cnPROCEED}.Q_{t'}$$

(iv) $t = e.d?v{::}t'$, $i \neq e_i$ for any $i = 1, \cdots k$:

$$\begin{aligned}P_i^t &= P_i^{t'}\\ Q_i^t &= Q_i^{t'}\end{aligned}$$

(v) $t = e_j.d?v{::}t'$

$$P_t^i = \begin{cases}(d!\langle v\rangle.\mathsf{DETECT.CHECK.PROCEED}.P_{t'}^i + \mathsf{LOCK}) + \mathsf{LOCK} & \text{if } i = j\\ & \\ \mathsf{CHECK.PROCEED}.P_{t'}^i & \text{otherwise}\end{cases}$$

$$Q_t = \mathsf{cnDETECT}(1).\mathsf{cnCHECK}.\mathsf{cnPROCEED}.Q_{t'}$$

B Virtual Shared Memory: Formal Definitions

In this Appendix we give the formal definitions needed to formalise both the implementation, and the specification, of the VSM application presented in §6.

In the following we let Locs be a set of memory locations, ranged over by $\mathsf{x}, \mathsf{y}, \cdots$. Requests are messages of the form $\mathbf{read}(\mathsf{x})$ or $\mathbf{write}(\mathsf{x}, n)$, ranged over by $r, r', \cdots$. the set of requests is denoted as Reqs. Lists of requests are ranged over by q, q', with ε denoting the empty list and $::$ denoting concatenation of symbols. Also, we assume a set of answers, ranged over $a, a', \cdots$, which are messages of the form $\mathsf{x} = n$. We let Answ be the set of answers and, with an abuse of notation, we use $q, q', \cdots$ to range over lists of answers. Finally, functions of the form $\sigma : \mathsf{Locs} \to \mathbb{N}$ represent memories. We let σ_0 be the initial memory, where $\sigma_0(\mathsf{x}) = \mathsf{0}$ for any $\mathsf{x} \in \mathsf{Locs}$.

The Specification. The specification consists of the system $\mathcal{V} = \Gamma_V \triangleright V$; the connectivity graph Γ_V is defined by letting $\Gamma_V \vdash cl_j \to i_j, i_j \to m, m \to o_j, o_j \to cl_j$, for $j = 1, 2$. This is a formal description of the connectivity graph depicted at Page 13. The system term V is defined as

$$V = i_1[\![\mathsf{In}_1]\!] \mid \mathsf{o}_1[\![\mathsf{Out}]\!] \mid \mathsf{m}[\![\mathsf{Mem}]\!] \mid \mathsf{o}_2[\![\mathsf{Out}]\!] \mid \mathsf{i}_2[\![\mathsf{In}_1]\!]$$

The processes $\mathsf{In_j}$, $\mathsf{Out_j}$, $j = 1, 2$, and Mem, will be defined shortly; at an intuitive level, $\mathsf{In_j}$ describes how requests are forwarded from cl_j to node m, $\mathsf{Out_j}$ how requests are forwarded from node m to cl_j, while Mem gives the implementation of a (centralised) memory, with a finite set of locations.

To handle the requests received by either node cl_j, the code for process $\mathsf{In_j}$ needs to store the received requests in a internal queue. To achieve this, we define a family of process definitions In_j^q, parameterised by a list of requests q. Intuitively this parameter captures the internal buffer of requests stored at some node. We let $\mathsf{In}_j = \mathsf{In}_j^{\varepsilon}$, for $j = 1, 2$, since we assume that initially there are no requests stored at nodes i_1, i_2. Let $j = 1, 2$; the process In_j^q is defined as $\mathsf{In_j^q} \Leftarrow \mathsf{Rec_j^q} + \mathsf{Fwd_j^q}$, where

$$\begin{aligned} \mathsf{Req_j^q} &\Leftarrow c?(r)\,.\mathsf{if}\ (r \in \mathsf{Reqs})\ \mathsf{then}\ \mathsf{In_j^{r::q}}\ \mathsf{else}\ \mathsf{In_j^q} \\ \mathsf{Fwd_j^{\varepsilon}} &\Leftarrow \mathbf{0} \\ \mathsf{Fwd_j^{q::r}} &\Leftarrow c_j!\langle r\rangle\,.\mathsf{In}_j^q \end{aligned}$$

Intuitively, process Req_j^q provides the capability to node i_j to parse messages, storing requests into the internal queue of node i_j, and ignoring all other messages. Process Fwd_j^q equips node i_j with the capability of forwarding the oldest stored request along channel c_j.

The behaviour of process Out_j is, in some sense, dual to the one of In_j. More specifically, at any given point process Out_j can receive an answer from node n, along channel d_j, or forward the next answer to node cl_j, along channel c.

Formally we let $\mathsf{Out}_j = \mathsf{Out}_j^{\varepsilon}$, and for any list of answers q we define

$$\begin{aligned}
\mathsf{Out}_{\mathsf{j}}^{\mathsf{q}} &\Leftarrow \mathsf{GetAnsw}_{\mathsf{j}}^{\mathsf{q}} + \mathsf{Reply}_{\mathsf{j}}^{\mathsf{q}} \\
\mathsf{GetAnsw}_{\mathsf{j}}^{\mathsf{q}} &\Leftarrow d_j?(a)\,.\mathsf{Out}_{\mathsf{j}}^{\mathsf{a::q}} \\
\mathsf{Reply}_{\mathsf{j}}^{\varepsilon} &\Leftarrow \mathbf{0} \\
\mathsf{Reply}_{\mathsf{j}}^{\mathsf{q::a}} &\Leftarrow c!\langle a\rangle\,.\mathsf{Out}_{\mathsf{j}}^{\mathsf{q}}
\end{aligned}$$

The code Mem placed at node m needs to equip the latter with a central memory σ. Further, it needs to define the code for processing requests forwarded by nodes i_1, i_2, and how replies to such operations are broadcast to nodes o_1, o_2. Formally, we define a family of processes $\mathsf{Mem}_{\sigma}^{\mathsf{q_1,q_2}}$, where σ is a memory and q_1, q_2 are two lists of requests, as follows:

$$\mathsf{Mem}_{\sigma}^{\mathsf{q_1,q_2}} \Leftarrow \mathsf{GetReq}_1^{\sigma,q_1,q_2} + \mathsf{GetReq}_2^{\sigma,q_1,q_2} + \mathsf{Exec}_1^{\sigma,q_1,q_2} + \mathsf{Exec}_2^{\sigma,q_1,q_2}$$

$$\begin{aligned}
\mathsf{GetReq}_1^{\sigma,\mathsf{q_1,q_2}} &\Leftarrow c_1?(r)\,.\mathsf{Mem}_{\sigma}^{(\mathsf{r::q_1}),\mathsf{q_2}} \\
\mathsf{GetReq}_2^{\sigma,\mathsf{q_1,q_2}} &\Leftarrow c_2?(r)\,.\mathsf{Mem}_{\sigma}^{\mathsf{q_1},(\mathsf{r::q_2})}
\end{aligned}$$

$$\begin{aligned}
\mathsf{Exec}_1^{\sigma,\varepsilon,\mathsf{q_2}} &\Leftarrow \mathbf{0} \\
\mathsf{Exec}_1^{\sigma,(\mathsf{q_1::\mathbf{read}(x)}),\mathsf{q_2}} &\Leftarrow d_1!\langle \mathsf{x} = \sigma(\mathsf{x})\rangle\,.\mathsf{Mem}_{\sigma}^{\mathsf{q_1,q_2}} \\
\mathsf{Exec}_1^{\sigma,(\mathsf{q_1::\mathbf{update}(x,n)}),\mathsf{q_2}} &\Leftarrow d_1!\langle \mathsf{x} = n\rangle\,.\mathsf{Mem}_{\sigma[\mathsf{x}\mapsto\mathsf{n}]}^{\mathsf{q_1,q_2}} \\
\mathsf{Exec}_2^{\sigma,\mathsf{q_1},\varepsilon} &\Leftarrow \mathbf{0}
\end{aligned}$$

$$\begin{aligned}
\mathsf{Exec}_2^{\sigma,\mathsf{q_1},(\mathsf{q_2::\mathbf{read}(x)})} &\Leftarrow d_2!\langle \mathsf{x} = \sigma(\mathsf{x})\rangle\,.\mathsf{Mem}_{\sigma}^{\mathsf{q_1,q_2}} \\
\mathsf{Exec}_2^{\sigma,\mathsf{q_1},(\mathsf{q_2::\mathbf{update}(x,n)})} &\Leftarrow d_2!\langle \mathsf{x} = n\rangle\,.\mathsf{Mem}_{\sigma[\mathsf{x}\mapsto\mathsf{n}]}^{\mathsf{q_1,q_2}}
\end{aligned}$$

We let $\mathsf{Mem} = \mathsf{Mem}_{\sigma_0}^{\varepsilon,\varepsilon}$, meaning that we initially assume that all memory locations are initialised to 0, and node m has not received any request to execute yet.

The Implementation. Here we assume an implementation of the VSM where a memory σ is partitioned into k nodes, $m_1, \cdots, m_k$. Let $\mathsf{Locs}_1, \cdots, \mathsf{Locs}_n \subseteq \mathsf{Locs}$ such that $\bigcup_{i=1}^{k} \mathsf{Locs}_i = \mathsf{Locs}$, and for any $i, j : 1 \leq i, j \leq k$, $\mathsf{Locs}_i \cap \mathsf{Locs}_j \neq \emptyset$ implies $i = j$. Intuitively, Locs_j represent the set of locations stored at node m_j. We let σ^j range over (partial) memories with domain Locs_j, that is $\sigma^j : \mathsf{Locs}_j \to \mathbb{N}$.

The implementation of the VSM protocol consists of the system $\mathcal{I} = \Gamma_I \triangleright I$. The connectivity graph Γ_I is defined by letting

$$\Gamma_I \vdash cl_j \to i_j, i_j \to lock, lock \to m_i, m_i \to unlock, unlock \to lock, lock \to o_j, o_j \to cl_j$$

for any $i = 1, \cdots, k$, $j = 1, 2$. This is a formal description of the connectivity graph depicted at Page 13.

The system term I is defined by letting

$$I = i_1[\![\mathsf{In_1}]\!]|\mathsf{i_2}[\![\mathsf{In_2}]\!]|\mathsf{lock}[\![\mathsf{Lock}]\!]|\left(\prod_{\mathsf{i=1}}^{\mathsf{k}} \mathsf{m_i}[\![\mathsf{Mem_i}]\!]\right)|\mathsf{unlock}[\![\mathsf{Unlock}]\!]|\mathsf{o_1}[\![\mathsf{Out_1}]\!]|\mathsf{o_2}[\![\mathsf{Out_2}]\!]$$

where $\mathsf{In_j}, \mathsf{Out_j}$, $j = 1, 2$ are the same processes used in the specification, while $\mathsf{Lock}, \mathsf{Unlock}$ and $\mathsf{Mem_i}$, $i = 1, \cdots, k$ are defined presently.

Let b be a flag which can assume either the value true or false. For any value of b, and list of requests q_1, q_2, we define the process $\mathsf{Lock_b^{q_1,q_2}}$ as follows:

$$\mathsf{Lock_b^{q_1,q_2}} \Leftarrow \mathsf{GetReqImpl}_1^{b,q_1,q_2} + \mathsf{GetReqImpl}_2^{b,q_1,q_2} + \mathsf{LockFwd}_b^{q_1,q_2}$$

$$\begin{aligned}
\mathsf{GetReqImpl_1^{b,q_1,q_2}} &\Leftarrow c_1?(r)\,.\mathsf{Lock_b^{(r::q_1),q_2}} \\
\mathsf{GetReqImpl_2^{b,q_1,q_2}} &\Leftarrow c_1?(r)\,.\mathsf{Lock_b^{q_1,(r::q_2)}}
\end{aligned}$$

$$\begin{aligned}
\mathsf{LockFwd}_{\mathsf{true},q_1,q_2} &\Leftarrow \mathsf{BroadReq}_1^{q_1,q_2} + \mathsf{BroadReq}_2^{q_1,q_2} \\
\mathsf{LockFwd}_{\mathsf{false},q_1,q_2} &\Leftarrow d_1?(a)\,.\mathsf{Lock_{true}^{q_1,q_2}} \mathsf{d_2?(a)}\,.\mathsf{Lock_{true}^{q_1,q_2}}
\end{aligned}$$

$$\begin{aligned}
\mathsf{BroadReq}_1^{\varepsilon,q_2} &\Leftarrow \mathbf{0} \\
\mathsf{BroadReq}_1^{(q_1::r),q_2} &\Leftarrow c_1!\langle r\rangle\,.\mathsf{Lock_{false}^{q_1,q_2}} \\
\mathsf{BroadReq}_2^{q_1,\varepsilon} &\Leftarrow \mathbf{0} \\
\mathsf{BroadReq}_2^{q_1,(q_2::r)} &\Leftarrow c_2!\langle r\rangle\,.\mathsf{Lock_{false}^{q_1,q_2}}
\end{aligned}$$

At any given time process $\mathsf{Lock_b^{q_1,q_2}}$ can receive a request along either channel c_1 or channel c_2; requests received along different channels are stored in different queues. The flag b regulated the capability of node *lock* to forward the requests stored in such queues to the memory nodes $m_1, \cdots, m_k$. If b is set to true then a request (either from the queue q_1 or from the queue q_2 can be broadcast, and b is set to false. This means that the memory nodes are now processing the request, and *lock* isl prevented from forwarding other requests until the current one has been executed. As we will see, when this happens node *unlock* broadcasts an answer along either channel d_1 or channel d_2. Upon receiving the answer, the ability of node *lock* to broadcast requests to memory nodes is restored, by setting value b to true. Initially we assume that *lock* can forward requests to the memory nodes, though its internal queues are empty. That is, we let $\mathsf{Lock} = \mathsf{Lock_{true}^{\varepsilon,\varepsilon}}$.

The code for the nodes $m_1, \cdots, m_k$ is relatively simple. Upon receiving a request, each node m_i, $i = 1, \cdots, k$ will check whether this corresponds to an operation involving a memory location stored in its own memory, in which case it will execute it and broadcast the answer to node unlock. Otherwise, the request is simply ignored by node m_i. Formally, for any request r, indexes $i =, 1, \cdots, k$ and $j = 1, 2$, and memory $\sigma_i : \mathsf{Locs}_i \to \mathbb{N}$, we define the collection of processes

$\mathsf{Mem}_{i,j}^{r,\sigma_i}$ as follows:

$$\mathsf{Mem}_{i,j}^{r,\sigma_i} \Leftarrow c_1?(r')\,.\mathsf{ExecImpl}_{i,1}^{r',\sigma_i} + c_2?(r')\,.\mathsf{ExecImpl}_{i,2}^{r',\sigma_i}$$

$$\mathsf{ExecImpl}_{i,j}^{\mathbf{read}(\mathsf{x}),\sigma_i} \Leftarrow \mathsf{if}\ (\mathsf{x} \in \mathsf{Locs}_i)\ \mathsf{then}\ c_j!\langle \mathsf{x} = \sigma_i(\mathsf{x})\rangle\,.\mathsf{Mem}_{i,j}^{r,\sigma_i}\ \mathsf{else}\ \mathsf{Mem}_{i,j}^{r,\sigma_i}$$
$$\mathsf{ExecImpl}_{i,j}^{\mathbf{write}(\mathsf{x},n),\sigma_i} \Leftarrow \mathsf{if}\ (\mathsf{x} \in \mathsf{Locs}_i)\ \mathsf{then}\ c_j!\langle \mathsf{x} = n\rangle\,.\mathsf{Mem}_{i,j}^{r,\sigma_i[\mathsf{x}\mapsto n]}\ \mathsf{else}\ \mathsf{Mem}_{i,j}^{r,\sigma_i}$$

Initially we assume that all locations in each memory node are initialised to 0, while the values of r and j are irrelevant. For any $i = 1, \cdots, k$, let $(\sigma_i)_0(\mathsf{x}) = 0$ for any $\mathsf{x} \in \mathsf{locs}_i$; then $\mathsf{Mem}_i = \mathsf{Mem}_{i,-}^{-,(\sigma_i)_0}$.

It remains to define process Unlock. This process only needs to forward the received answer along the appropriate channel (that is, by changing c_j to d_j, for $j = 1, 2$). That is,

$$\mathsf{Unlock} \Leftarrow (c_1?(a)\,.d_1!\langle a\rangle\,.\mathsf{Unlock}) + c_2?(a)\,.d_2!\langle a\rangle\,.\mathsf{Unlock}).$$