Interfaces in Ecosystems: Concepts, Form, and Implementation

Manuela Bujorianu[1]0000-0001-9630-1868, Tristan Caulfield[1]0000-0002-7039-6472, Marius-Constantin Ilau[1]0000-0002-5424-5375, and David Pym[1,2,3]0000-0002-6504-5838

[1] Department of Computer Science, University College London
[2] Department of Philosophy, University College London
[3] Institute of Philosophy, University of London
{l.bujorianu,t.caulfield,marius-constantin.ilau.18,d.pym}@ucl.ac.uk
david.pym@sas.ac.uk

Abstract. The world we inhabit is constantly increasing in complexity and with this comes a need for new ways of understanding and thinking that can manage this complexity. A key part of managing this complexity involves constructing models of ecosystems of systems and reasoning about their properties. One approach to modelling systems has been based on a process-algebraic formulation of an abstract view of distributed systems. Here, building on this view, we concentrate on modelling ecosystems of systems by developing a rather general concept of interface between systems. Interfaces between models describe how they can be composed together to construct a model of an ecosystem. By introducing the idea of local reasoning, building on work in program analysis and verification, we develop tools for analysing the behaviour and performance of ecosystem models in an efficient way.

Keywords: Ecosystems · Interfaces · Modelling

1 Introduction

The concept of a distributed system is a basic one in informatics, deriving from core ideas in the theory of computer systems. Traditionally, this would mean a collection of computing components which appear as a singular, coherent entity, can communicate and have the same notion of time. In the context of ecosystems of systems — now pervasive in our world — this serves as a starting point for a consistent modelling theory explicitly considering heterogeneity. The argument is as follows: to deal with the variety of existing systems and their increasing complexity from a modelling perspective, a decomposition to a set of basic concepts must be constructed; this set of concepts has to be general enough to allow for modelling ecosystems, but at the same time be simple enough not to alter or over-complicate the model meaning. We abstract this set of concepts from the notion of distributed system and construct what we call, by generalization, the *distributed systems metaphor*.

For the practical purposes of simulation modelling — see, for example, [52] for a survey of the overall ideas — the executions of models can be captured, for instance, by the dynamics of the many forms of process algebra. Examples include [34, 25, 13, 14, 12, 2, 11, 9, 24] and many more, including those, described in various previous Simutools papers [14, 11, 9] that are intended to support implementations of the 'distributed systems metaphor' — in which a system is modelled in terms of its *locations*, *resources*, and *processes*, relative to its *environment* — as described in [13, 14, 12, 2, 11, 9]. The history of this work can be traced to Birtwistle's 'Demos: Discrete Event Modelling on Simula' [7]. These approaches can be implemented by tools such as those described in languages such as Gnosis [14, 12] and Julia [6, 28]. In particular, the modelling approach based on the 'distributed systems metaphor', is conveniently captured in Julia using the 'SysModels' package that is available at [8]. A more abstract view may be found in [20] (in these proceedings).

The distributed systems metaphor notwithstanding, there are very many — far too numerous to discuss here — informal definitions of systems discussed across many disciplines. However, following [20], we can identify a common core of these definitions, all supported by the distributed systems metaphor and its implementations, that will serve as a starting point for our work:

> A system is something whose behaviour is reflected in the behaviour of its components.

This definition is deceptively simple: it identifies, at least implicitly, not only *behaviour* and *components*, but also *composition* (necessarily, of components) and so (again, necessarily) *interfaces* between components. This definition amounts to an abstraction and generalization of the characterization of systems that is provided by the distributed systems metaphor (e.g., [10, 9]).

In this paper, we work towards constructing a framework and discipline for model engineering that will provide — in a scale-free form, applicable across the scope of the distributed systems metaphor — the tools and approaches required for building models of large, complicated ecosystems. We start from building basic models using the distributed systems paradigm before introducing a general concept of interface and discussing how such models can be composed using these. Beyond the structural requirements for model composition, we also introduce the idea of dynamic model checking, to verify that the behaviour of composed models is correct with respect to the requirements of their interfaces.

To understand the role of dynamic model checking, it is perhaps helpful to compare with the idea of run-time verification from computer science. The main difference between runtime verification and the approach presented in this paper can be observed by understanding the nature of the object of inquiry: the former interacts with a system, whereas the latter interacts with a system representation. Runtime verification is a procedure applied at the level of a software system, usually based on a different set of abstractions than those used in the system's design and development. In other words, it is an external procedure that is used for monitoring properties of system states directly. Although they have similar goals, dynamic model checking, as a procedure, operates at a different conceptual layer, serving as a verification tool for model properties that have already been abstracted or inferred from a system. Thus, in run-time verification, the abstraction is implicit and mainly focused on the definition of properties, whereas in dynamic model checking, the abstraction and inference processes used for designing and constructing the model and the definition of model properties are underlined by the same explicit distributed systems metaphor. The benefits are twofold: firstly, conceptual consistency across model design, construction, and verification is required to limit possible errors in translating the model properties to system properties; secondly, this implies a higher degree of generality for dynamic model checking.

In Section 2, we discuss our use of distributed systems as a metaphor for ecosystems of systems. We discuss how to model distributed systems at the level of abstraction of the metaphor, including giving a presentation that is adapted to the needs of implementation, and explain how to associate a rich logical language with such models. In Section 3, we explain how to model the compositional structure of ecosystem models using a rather general notion of interface for the implementation-adapted presentation version of models together some logical specification of required properties and policies. Then, in Section 4, we explain how our theory of composition and local reasoning can be used to support reasoning about the behaviour of composite models. Particularly, we show that our notion of locality includes meaning explicitly and is not solely grounded on model state: to reason about the property of a sub-model locally implies using only the relevant and meaningful information, even if the model state contains more than that. In Section 5, we explain how to establish a theory of local reasoning (in the sense of Separation Logic [27, 46]) for our interface-mediated theory of composition and discuss how it supports a substitution operation for models. Appendices A and B sketch the underlying process-theoretic and logical theory.

Related work, including [51, 50, 16, 17] among others, also addresses the concept of interface, but lacks the close structural integration with logic and does not address the ideas around local reasoning. Milner's work on bigraphs considers interfaces in an abstract setting (e.g., [37]), and has been related to logical work (e.g., [15]), but again local reasoning is not considered. We give a more complete account of related work on interfaces in Appendix C, which explains how our very general, yet ecosystem-motivated, notion of interface captures a wide range of the notions of interface that are in the literature on systems and modelling.

2 The Distributed Systems Metaphor

We use the distributed systems metaphor to describe ecosystems. We begin, in this section, with how the metaphor describes individual systems. Our modelling approach for ecosystems is then component-based. Ecosystems are thought of as systems of systems, or systems with different components, which interact, collaborate, inter-operate to achieve coherent goals.

In moving from modelling systems to ecosystems of systems, it is therefore necessary to define interfaces that enable interactions between component systems. Such interfaces will be the focus of this paper and are developed and analysed in subsequent sections.

For now, beginning with the motivation of capturing individual systems, there are three key ingredients upon which we draw:

- *Location.* Distributed systems naturally have a concept of location and distinct locations may be connected to one another. This is the system's underlying structure or architecture. In the setting of computer systems, components are present at different locations and connected by a network. In the more general view, locations can be physical (e.g., a room, a container), logical (e.g., an address in computer memory), or abstract (e.g., the location where a semaphore exists).
- *Resource.* Resources exist at locations and are manipulated by processes as they execute. Examples of resources include units of data (and information), physical entities, and other derived notions.
- *Process.* Processes execute and manipulate resources — such as consuming, creating, and moving them between locations — as they execute and deliver the required system behaviour.

These concepts can be used to build a representation of a system's structure and operation, but there is one more concept required: the environment in which the system operates.

- *Environment.* Environments capture the world outside of, and its interaction with, the system of interest.

Throughout this paper we illustrate its ideas and approach with a recurring example of supply chains — classically, the sequence of processes involved in the production and distribution of a commodity, but now also encompassing information supply — with our basic conception of the structure of supply chains being essentially that of [22]. In this view, can be seen to be a quite generic notion of system. The supply chain example will be used in more detail in subsequent sections. For now, we focus on more generic issues.

Figure 1 illustrates the general set-up that we seek to analyse. Here the idea is that two supply chains, the input chain and the output chain, are connected by an interface. From one point of view, each of the input and output chains is itself an ecosystem of component systems, respectively C_is and D_js. From another, each of the input and output ecosystems can be seen as a single distributed system. Either way, the range of architectures of supply chains — encompassing forms of serial and parallel composition — can be described in the ways given in [22]. This illustrates that the distinction between what constitutes the component systems and what constitutes is a design decision that should be driven by an understanding of the purpose of the system.

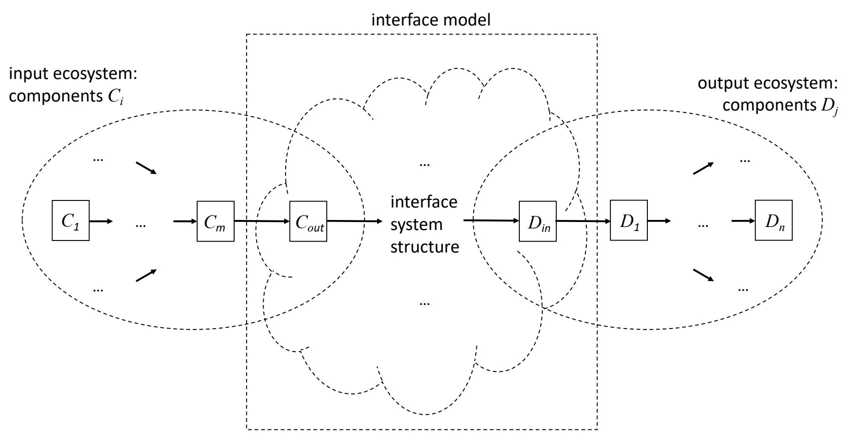
interface model
input ecosystem: components C_i
output ecosystem: components D_j
C_1
C_m
C_{out}
interface system structure
D_{in}
D_1
D_n

Fig. 1. A composition of supply chains

From the ecosystem perspective, each of the component systems (i.e., the Cs and Ds) resides at a location. These locations are connected by the routes along which the commodities supplied to and from the components are moved. Of particular importance in this view is the role of the interface, which may in general itself be modelled as system in the same way as the components. In order for the input and output ecosystems to be composed, two key conditions must be satisfied: first, each of the input and output ecosystems must have an appropriate architecture of locations — that is, their graphs must 'match' with the interface mode at C_{out} and D_{in} — and their modification functions (that is, the signatures of the models) must be consistent; second, there must be satisfaction of a range of logically expressible properties — for example, that, at a given point in the execution, sufficient quantities of a given resource are available at a specified location or that all of a given collection of supplying components have completed their required supply tasks.

Checking the properties of large complex systems for compatibility is, in general, an expensive, perhaps intractable, task. In this paper, as described in subsequent sections, we propose a theory and implementation of local reasoning in which we check, dynamically, the properties that are required to ensure that composition through given interfaces works as intended. In practice, this will capture most of the possible sources of error and, in particular, support checking that the substitution of one component for another preserves the required functionality of the ecosystem.

2.1 The Mathematics of Modelling Distributed Systems

The formal mathematical framework that captures the distributed systems metaphor for modelling systems is presented in Appendices A and B. In Appendix A, we given a calculus of processes in which location-resource-process states co-evolve as actions occur. This is written as $L, R, E \xrightarrow{a} L', R', E'$, which the action a evolves the state L, R, E — read as the process state E evolving at location L relative to resources R at location L — to a new state L', R', E'. The three components are defined separately. The pure process part is essentially Milner's SCCS [32], the resource part is based on the semantics of BI [42, 44], the logic that underpins Separation Logic [27, 46], and locations and their connections are captured by directed graphs.

These components naturally fit together in the presence of a 'modification function' from locations, resources, and (basic) actions to locations and resources that describes the basic signature of the model. Moreover, all of these structures are in a sense primitive: Milner's SCCS is perhaps the most primitive process algebra that is complete for describing recursively enumerable graphs and employs a synchronous concurrent product, from which, under modest assumptions, asynchronous products can be recovered (but not vice versa) [32, 18]. This latter property is convenient in the modelling context: semantically, we can consider a model to be given by collections of locations and resources together with a modification function that specifies the actions that are available to construct processes [13, 12, 2].

Associated with the calculus of processes is a substructural logic, described in Appendix B, that has natural interpretation as a logic of resources (and also locations) in the sense of BI [42, 44] and Separation Logic [27, 46]. Properties ϕ of states L, R, E are described by a semantic judgement of the form $L, R, E \models \phi$ and natural notions of process equivalence and logical equivalence coincide (see Appendix B). These correspondences lead to a natural and conceptually simple modelling framework.

In the basic presentation of the mathematical set-up, we can suppress the concept of location, though not of course in the implemented models described in Section 2.2. Technically, locations are handled in essentially the same way as resources, and suppressing them simplifies the presentation of the basic theory. The framework we sketch in Appendices A and B has been developed in detail in [13, 12, 10, 2].

2.2 Implementing Models of Distributed Systems

As we have discussed, an ecosystem is thought of as a system of systems that brings together a collection of systems in order to achieve and overall goal. The composite supply chain, as in Figure 1, Each constituent system may have its own policies, objectives, and resources while coordinating within the whole ecosystem and adapting to accomplish the overall goals.

The description of models sketched above, in Section 2, lies at a wholly semantic level and is not formulated with the needs of implementation foremost.

Our implementations — in Julia [28] and building on an earlier bespoke tool called Gnosis [14, 12], using the SysModels package [48] — are based on the more specific formulation of *basic models*, building on the ideas presented in [10], which we now sketch and which are the basis of our implementation work.

A basic model is intended to capture the simplest convenient representation of a model of a single system using the distributed systems metaphor. Each basic model encapsulates the primitives we have discussed above — locations, resources, processes, and environment. In Section 3, we give an account of interaction between models through a notion of interface formulated at a similar level of abstraction.

Basic Model. A basic model is a representation of a single system that corresponds to the semantic definition discussed above, but which specifies locations as directed graphs, with vertices labelled by resources, and which identifies the actions and processes that are available.

Formally, a basic model is a tuple $M = \langle \mathcal{G}(\mathcal{V}[\mathcal{R}], \mathcal{E}), \mathcal{A} \rangle$, where $\mathcal{G}(\mathcal{V}[\mathcal{R}], \mathcal{E})$ is a directed graph with vertices $\mathcal{V}$, labelled with resources $\mathcal{R}$, and edges $\mathcal{E}$, and $\mathcal{A}$ is a set of actions specified by a modification function μ — given by $\mu : (a, \mathcal{V}, \mathcal{R}) \mapsto (\mathcal{V}', \mathcal{R}')$, the action a evolves the process E to E', $\mathcal{R}$ to $\mathcal{R}'$, and $\mathcal{V}$ to $\mathcal{V}'$ — as defined in Appendix A.

It is this specification that is used as the basis of the Julia implementation. Locations are specified as objects with links to other location objects — forming a (directed) graph. These locations can contain resources, which are specified using Julia types. Processes are implemented as Julia functions, which provides a natural way of expressing sequences of actions (such as moving resources, or waiting for some duration, and so on).

A scheduler coordinates the execution of these processes. Models are executed on a single thread, but in a way that simulates parallel execution of processes. Events — parts of processes to be executed — are stored in a queue, ordered by time; the scheduler dequeues the first event and executes it. For example, a process that is currently executing at time t can pause its execution for 30 seconds. It gets paused and added to the queue to resume execution at time $t + 30$, and the scheduler begins executing the next item from the queue. In [14, 12], the scheduler for Gnosis is given a semantics in terms of the calculus of locations, resources, and processes discussed above. The scheduler implemented in Julia could be given a very similar semantics.

The implementation also handles contention for resources. Processes must claim resources before they use them, and can release them after. Only one process can claim a particular resource at a time. If a desired resource is not available (because it has already been claimed, or because it is not present) a process will wait until the resources become available, so forming an implicit queue for resources.

The implementation of these concepts provides a useful tool for building executable models of systems. In the next sections, we explore how the concepts and implementation can be augmented to capture interactions between systems, which forms the basis for modelling ecosystems.

3 Interfaces and Composition

To begin thinking about how systems interact, we first need a concept that captures where and how these interactions happen. In this section, we introduce the *basic interface* and then extend our definition of basic model with this concept.

It is helpful to first revisit the concept of environment. Systems do not exist in isolation; they interact with the world — other systems — around them. However, we do not always want to explicitly model all of these adjacent systems. Instead, we use the concept of an environment, which models external events that have an effect on the system. When these events occur, associated actions inside a model are executed.

In the distributed systems paradigm, model-model interaction can happen in two ways. First, akin to environment-model interaction, one model can initiate an action in another model. Second, models might share locations and interact by manipulating resources in these shared locations. The combination of both of these is also possible: one model may move a resource into a shared location and initiate an action, for example.

Basic Interfaces. A *basic interface*, which describes the locations and actions involved in such interactions, is defined as follows: $I = \langle \mathcal{G}(\mathcal{V}_I[\mathcal{R}_I], \mathcal{E}_I), \mathcal{A}_I \rangle$, where $\mathcal{G}(\mathcal{V}_I[\mathcal{R}_I], \mathcal{E}_I)$ is a directed graph with vertices $\mathcal{V}_I$, labelled with resources $\mathcal{R}_I$, and edges $\mathcal{E}_I$, and $\mathcal{A}_I$ is a set of actions specified by a modification function.

Model. The locations and actions in a basic interface lack purpose unless associated with a model. A *model* is given by a tuple $M = \langle \mathcal{G}(\mathcal{V}[\mathcal{R}], \mathcal{E}), \mathcal{A}, \mathcal{I} \rangle$, where $\langle \mathcal{G}(\mathcal{V}[\mathcal{R}], \mathcal{E}), \mathcal{A} \rangle$ is a basic model, and $\mathcal{I}$ is a set of basic interfaces such that $\mathcal{G}(\mathcal{V}_I[\mathcal{R}_I], \mathcal{E}_I)$ is sub-labelled graph of $\mathcal{G}(\mathcal{V}[\mathcal{R}], \mathcal{E})$ (that is, it is a subgraph and all the resources labelling each vertex in the interface also label the same vertex in the model) and $\mathcal{A}_I \subseteq \mathcal{A}$.

Environment. We can now (informally) define an *environment* of a model. It associates to some or all of the actions in a model's basic interfaces a probability distribution. In an implementation of this framework, actions that are associated with probability distributions are initiated by sampling the distribution.

Composition of Models. A model as defined above can have multiple basic interfaces, each defining parts of the model — locations and actions — through which it can interact with other models. The *composition* operation joins two models using a specified basic interface from each one.

Let $M_1 = \langle \mathcal{G}(\mathcal{V}_1[\mathcal{R}_1], \mathcal{E}_1), \mathcal{A}_1, \mathcal{I}_1 \rangle$ and $M_2 = \langle \mathcal{G}(\mathcal{V}_2[\mathcal{R}_2], \mathcal{E}_2), \mathcal{A}_2, \mathcal{I}_2 \rangle$ be two models whose sets of actions have consistent modification functions (i.e., that agree on actions present in both). Consider two basic interfaces

$$I_1 = \langle \mathcal{G}(\mathcal{V}_{I_1}[\mathcal{R}_{I_1}], \mathcal{E}_{I_1}), \mathcal{A}_{I_1} \rangle \text{ and } I_2 = \langle \mathcal{G}(\mathcal{V}_{I_2}[\mathcal{R}_{I_2}], \mathcal{E}_{I_2}), \mathcal{A}_{I_2} \rangle$$

and suppose that at least one of $\mathcal{V}_{I_1} \cap \mathcal{V}_{I_2} \neq \emptyset$ and $\mathcal{A}_{I_1} \cap \mathcal{A}_{I_2} \neq \emptyset$. The composition $M_1 \circ_{I_1,I_2} M_2$, where $I_1 \in \mathcal{I}_1$ and $I_2 \in \mathcal{I}_2$, is defined pleasingly simply as follows:

$$M_1 \circ_{I_1,I_2} M_2 = \langle \mathcal{G}(\mathcal{V}_1 \oplus \mathcal{V}_2[\mathcal{R}_1 \oplus \mathcal{R}_2], \mathcal{E}_1 \oplus \mathcal{E}_2), \mathcal{A}_1 \oplus \mathcal{A}_2, \mathcal{I}_1 \oplus \mathcal{I}_2 \rangle$$

where $\mathcal{V}_1 \oplus \mathcal{V}_2 = \mathcal{V}_1 \cup \mathcal{V}_2$, $\mathcal{E}_1 \oplus \mathcal{E}_2 = \mathcal{E}_1 \cup \mathcal{E}_2$, $\mathcal{A}_1 \oplus \mathcal{A}_2 = \mathcal{A}_1 \cup \mathcal{A}_2$, $\mathcal{I}_1 \cup \mathcal{I}_2$, and for each $v \in \mathcal{V}_1 \oplus \mathcal{V}_2$,

$$v[\mathcal{R}_1 \oplus \mathcal{R}_2] = \begin{cases} v[\mathcal{R}_1] & = \text{ if } v \in \mathcal{V}_1 \wedge v \notin \mathcal{V}_2 \\ v[\mathcal{R}_2] & = \text{ if } v \in \mathcal{V}_2 \wedge v \notin \mathcal{V}_1 \\ v[\mathcal{R}_1 \cup \mathcal{R}_2] & = \text{ otherwise} \end{cases}$$

In this definition, it might be expected that the interfaces used to form a composition would not be available for further compositions, so that we should then have $\mathcal{I}_1 \cup \mathcal{I}_2 \backslash \{I_1, I_2\}$ above. However, such a requirement is not needed and its absence provides a more general and practically more realistic model of the concept of interface. The requirement that at least one of $\mathcal{V}_{I_1} \cap \mathcal{V}_{I_2} \neq \emptyset$ and $\mathcal{A}_{I_1} \cap \mathcal{A}_{I_2} \neq \emptyset$ ensures that we are actually achieving composition. If both intersections are empty, then the two models are necessarily executing independently of each other.

The following basic properties are easy to establish:

Proposition 1 (Composition Soundness). *If M_1 and M_2 are models as above, then so is $M_1 \circ_{I_1,I_2} M_2$.*

Proposition 2 (Commutativity). *If M_1 and M_2 are models as above, then $M_1 \circ_{I_1,I_2} M_2 = M_2 \circ_{I_2,I_1} M_1$.*

Proposition 3 (Associativity). *Consider the compositions $(M \circ_{I_1,I_2} N) \circ_{I_3,I_4} P$ and $M \circ_{J_1.J_2} (N \circ_{J_3,J_4} P)$ of models M, N, and P using interfaces I_is and J_js, for $1 \leq i, j \leq 4$. Then*

$$(M \circ_{I_1,I_2} N) \circ_{I_3,I_4} P = M \circ_{J_1.J_2} (N \circ_{J_3,J_4} P)$$

provided, for $1 \leq k \leq 4$, $I_k = J_k$.

When two models are composed using their basic interfaces it follows, by construction, that their basic architectures fit together. This, however, is not enough for their composition necessarily to function as intended. Each model has a collection of functions that it is intended to be able to perform and a collection of properties that can be expected of it and, whenever it is composed with another model, it is necessary to specify which of these must be preserved.

The simplest away to achieve this is to associate with each model a logical formula, using the logical language of Appendix B, that expresses the functionality and properties that must be preserved.

With this last step, we at last have a viable notion of interface. This notion substantially generalizes and makes cleaner that given in [10].

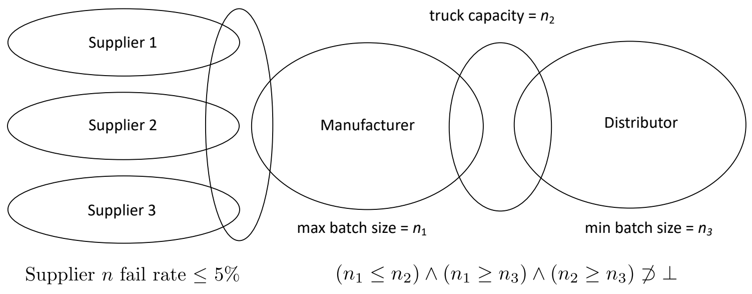

Fig. 2. Composition of Supply Chains: Structure and Logical Conditions

Interface Model. An interface model is a pair $I = \langle M, \phi \rangle$, where M is a model, and ϕ is a set of formulae (in the language of logic described in Appendix B) that describe properties of the model that must be preserved under composition. We identify a set of formulae with the conjunction ($\wedge$) of its elements as necessary. We refer to these formulae as *interface formulae*.

Note that the notion of an interface model strictly generalizes that of a model: simply take the interface model $\langle M, \{\top\} \rangle$, which places no constraint on M.

When two interface models are composed, it is necessary that their basic structure (that is, their basic actions do the same things at the same places) and the properties that they each require to be preserved under composition be consistent. These requirements are captured by the notion of admissibility, which requires that their modification functions (signatures) and their interface formulae be consistent.

Admissibility. Let $\langle M_1, \phi_1 \rangle$ and $\langle M_2, \phi_2 \rangle$ be interface models such that M_1 and M_2 have consistent modification functions (i.e., that agree on actions present in both). Let $M_1 \circ M_2$ denote a composition of M_1 and M_2 using some choice of interfaces. Then the composition of $\langle M_1, \phi_1 \rangle$ and $\langle M_2, \phi_2 \rangle$, denoted $\langle M_1, \phi_1 \rangle \circ \langle M_2, \phi_2 \rangle := \langle M_1 \circ M_2, \phi_1 \wedge \phi_2 \rangle$, is admissible if $\phi_1 \wedge \phi_2 \not\supset \perp$.

A simple example of this is illustrated in Figure 2 in which the composition of the suppliers, the manufacturer, and the distributor is supported bu interfaces that match the architecture and in which the necessary conditions are all consistent.

Proposition 4 (Composition Soundness). *If $\langle M_1, \phi_1 \rangle$ and $\langle M_2, \phi_2 \rangle$ are interface models, then so is their composition $\langle M_1, \phi_1 \rangle \circ \langle M_2, \phi_2 \rangle$.*

Proposition 5 (Commutativity and Associativity). *Commutativity of composition of interface model follows as for models. Associativity of composition of interface models $\langle M_1, \phi_1 \rangle$, $\langle M_2, \phi_2 \rangle$, and $\langle M_3, \phi_3 \rangle$ holds as for models provided also $\phi_1 \wedge \phi_2 \wedge \phi_3 \not\supset \perp$.*

The notion of composition established in this section determines the conditions under which, in a model of an ecosystem, one component model may be replaced by another of the same component system. The concept of substitution is developed in Section 5.

4 Dynamic Model Checking

We have built up a picture of how ecosystem models can be constructed, building on the concepts of basic model and basic interface to define composition of models, finishing with the definition of interface models, which associate logical formulae with models. These formulae express the required functionality and properties of models that must must be preserved, particularly when models are composed. This is important for a variety of purposes. For example, it allows the modeller to ensure: that model behaviour matches the real-world behaviour of the system being modelled; that the composition of models is correct and matches specification and behaves as intended; and, that changes in model parameters or substitution of models still satisfy requirements. We propose *dynamic model checking* as a method of verifying these formulae during the execution of implemented models. This is different from the usual notion of model checking (e.g., [3, 21]): dynamic model checking verifies a model during the circumstances of a particular execution of a model, rather than under all possible circumstances; it is localized in space and time — it checks properties only in a small region of the model, at the interfaces, at specific times during execution.

If there are stochastic elements in the model, the results of dynamic model checking may change depending on what happens during the execution of a model. Repeated executions of the model are required to reach a desired level of confidence that model is correct with respect to the specified formulae.

To implement dynamic model checking using the Julia SysModels package [28, 48], we need a way to express resources, locations, process execution, and formulae, together with the ability to evaluate formulae as the model executes. This follows the pattern introduced in Section 2.2:

- resources residing at locations are represented in the code of an implemented model as a vector of resource objects, declared as Julia types, inside a location object, with links to other objects;
- process evolution, through occurrences of sequences of actions, is represented using Julia functions;
- *then* interface formulae are represented as Julia functions that return booleans, based on the state of processes and the presence of resources in locations.

We illustrate this using the simple supply-chain example shown in Figure 2. In this scenario, a number of suppliers supply goods to a manufacturer, which transforms them into finished goods and transfers them to a distributor. For this example, we consider two properties that we want to verify using dynamic model checking. The first is the failure rate of the goods from the suppliers, and the second is the compatibility between the manufacturer's and distributor's batch sizes and the capacity of the trucks that is used to move the finished goods between them.

The manufacturer requires a failure rate of less than 5% from the suppliers. We can express this in the logical language described in Appendix B as $\phi = (\text{Fail(Supplier n)} \leq 5\%)$, for each Supplier 1, 2, 3. This needs to be translated into a statement in the Julia language to work with the implementation:

```
state(proc_goods, ‘receiving’) &&
length(at(loc_input, r-> r isa FaultyGood)) /
length(at(loc_input, r-> r isa InputGood))
< 0.05
```

Formulae are implemented as functions that return true/false; they check the state of processes using the `state` function and can inspect resources at locations using `at`. Processes use a pair of functions `set_state` and `reset_-state` to declare when they enter and leave a particular state — essentially giving a label to an action or sequence of actions for reference by formulae. The `at` function can specify a filter, used to select only certain types of resource or resources with specific properties from a location.

This example is checking whether the `proc_goods` process is in a particular state (‘receiving’) and whether the ratio of faulty goods to correct goods in location `loc_input` is less than 5%, using `at` to select `FaultyGood` resources and then all `InputGood` resources. The `length` function is built into Julia and returns the size of the vector.

For efficiency, formulae are only evaluated when relevant locations or process states are updated. Here, formulae are evaluated only when resources are added or removed at `loc_input`, or when the state of the process is updated to ‘receiving’. We can write formulae to check that batch sizes are correct,

```
state(distrib_proc, ‘receiving’) &&
length(at(loc_distributor, r->r isa FinishedGood)) >=
distrib_min_size
```

and also that truck capacity is not exceeded,

```
state(truck_proc, ‘collecting’) &&
length(at(loc_output, r->r isa FinishedGood)) <=
truck_max_size
```

These formulae are associated with an interface model in the code. The first formula above is associated with the manufacturer model, the second with the distributor model, and the final with the trucking model. All the models are composed to form one model.

Although the scenario, models, and formulae in this example are fairly trivial, they demonstrate how all of the concepts we have discussed in this paper come together to enable the construction and the checking of composition in ecosystem models. Because models are built using resource, location, and process, the relatively simple `state` and `at` used in the construction of formulae can be used to build up complex expressions about the state of a model. Future work will increase the expressiveness of the implemented logic; for example, referring to a history of states.

5 Model Management: Substitution and Local Reasoning

In this section, we discuss how substitution and local reasoning, two tools which support the management of the modelling process, are enabled with the approach presented above. The practice of modelling is active: as models of ecosystems are developed, it is natural that the function and specification of their component models may evolve. If one component model is replaced withe another, we must ensure that the resulting model still satisfies the requirements for it to interact with the other models in the ecosystem.

For example, consider the supply chain depicted in Figure 2. Suppose Supplier 2 ceases to trade and must be replaced by a different supplier. We can substitute a model of the new supplier. The conditions for the Manufacturer remain the same, and we can verify that the modified supply chain, including the new supplier, continues to satisfy the structural requirements, as specified by the modification function, and the logical condition on the fail rate.

It can be seen that the formulation of the concepts of interface and dynamic model checking presented here support this requirement for substitution. Indeed, dynamic model checking can be seen as delivering a form of local reasoning, as has been very successfully employed in program analysis and verification [27, 46, 30], but at the level of general ecosystem models.

6 Conclusion: Towards Model Engineering

In this paper, we have presented an approach to modelling ecosystems based on concepts from distributed systems. We have developed the notion of a basic model, expanded it with basic interfaces to define a model, and and subsequently to an interface model, which has associated formulae that characterize the requirements of composition. We have then described how these formulae can be evaluated as a model executes, in order to verify properties of the model, through dynamic model checking.

This approach enables the composition of models in such a way that properties essential for the composition to function properly can be verified locally at the interfaces between components, while avoiding the need to verify the correctness of components while doing so. This supports more robust construction of large ecosystem models at limited computational cost. It also provides support for useful modelling tools such as substitution and principles of local reasoning that go beyond the basic requirements of composition. We view this work as early progress towards a rigorous discipline of model engineering.

References

1. Luca Alfaro and Thomas Henzinger. Interface automata. *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 26, 09 2001.
2. G. Anderson and D. Pym. A calculus and logic of bunched resources and processes. *Theoretical Computer Science*, 614:63–96, 2016.
3. C. Baier and J.-P. Katoen. *Principles of Model Checking*. MIT Press, 2008.
4. Albert Benveniste, Benoît Caillaud, Alberto Ferrari, Leonardo Mangeruca, Roberto Passerone, and Christos Sofronis. Multiple viewpoint contract-based specification and design. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *Formal Methods for Components and Objects*, pages 200–225, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.
5. Albert Benveniste, Benoît Caillaud, Dejan Nickovic, Roberto Passerone, Jean-Baptiste Raclet, Philipp Reinkemeier, Alberto Sangiovanni-Vincentelli, Werner Damm, Thomas A. Henzinger, and Kim G. Larsen. Contracts for system design. *Foundations and Trends® in Electronic Design Automation*, 12(2-3):124–400, 2018.
6. Jeff Bezanson, Stefan Karpinski, Viral B. Shah, and Alan Edelman. Julia: A fast dynamic language for technical computing. 2012. `arXiv:1209.5145`.
7. G. Birtwistle. *Demos — discrete event modelling on Simula*. Macmillan, 1979.
8. T. Caulfield. SysModels Julia Package. Available at Available at `https://github.com/tristanc/SysModels`. Accessed 10/05/2021.
9. T. Caulfield, M.-C. Ilau, and D. Pym. Engineering ecosystem models: Semantics and pragmatics. In *International Conference on Simulation Tools and Techniques*, pages 236–258. Springer, 2022.
10. T. Caulfield and D. Pym. Modelling and simulating systems security policy. In *Proc. SimuTools*, 2015.
11. Tristan Caulfield and David Pym. Modelling and simulating systems security policy. *EAI Endorsed Transactions on Security and Safety (Proc. Simutools 2016, Prague)*, 3(8):e3–e3, 2016.
12. M. Collinson, B. Monahan, and D. Pym. *A Discipline of Mathematical Systems Modelling*. College Publications, 2012.
13. M. Collinson and D. Pym. Algebra and logic for resource-based systems modelling. *Math. Structures in Comput. Sci.*, 19:959–1027, 2009.
14. Matthew Collinson, Brian Monahan, and David Pym. Semantics for structured systems modelling and simulation. In *Proc. Simutools 2010*. ACM Digital Library, ISBN 78-963-9799-87-5, 2010.
15. Giovanni Conforti, Damiano Macedonio, and Vladimiro Sassone. Spatial logics for bigraphs. In Luís Caires, Giuseppe F. Italiano, Luís Monteiro, Catuscia Palamidessi, and Moti Yung, editors, *Automata, Languages and Programming*, pages 766–778, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.
16. Luca de Alfaro and Thomas A. Henzinger. Interface automata. New York, NY, USA, 2001. Association for Computing Machinery.
17. Luca de Alfaro and Thomas A. Henzinger. Interface theories for component-based design. In Thomas A. Henzinger and Christoph M. Kirsch, editors, *Embedded Software*, pages 148–165, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.
18. R. de Simone. Higher-level synchronising devices in Meije-SCCS. *TCS*, 37:245–267, 1985.
19. D. Galmiche, D. Méry, and D. Pym. The Semantics of BI and Resource Tableaux. *Math. Structures in Comput. Sci.*, 15:1033–1088, 2005.

20. Galmiche, Didier and Lang, Timo and Pym, David. Minimalistic System Modelling: Behaviours, Interfaces, and Local Reasoning. In *Proc. 16th EAI International Conference on Simulation Tools and Techniques (SIMUtools)*, Berlin, Heidelberg, 2024. Springer Berlin Heidelberg. Preprint: `https://arxiv.org/abs/2401.16109`.
21. O. Goudsmid, O. Grumberg, and S. Sheinvald. Compositional model checking for multi-properties. In F. Henglein, S. Shoham, and Y. Vizel, editors, *Verif., Mod. Check., and Abs. Int.*, pages 55–80. Springer, 2021.
22. H. Håkansson and G. Persson. Supply chain management: The logic of supply chains and networks. *The International Journal of Logistics Management*, 15(1):11–26, 2004.
23. M. Hennessy and G. Plotkin. On observing nondeterminism and concurrency. In *Proceedings of the 7th ICALP*, volume 85 of *Lecture Notes in Computer Science*, pages 299–309. Springer-Verlag, 1980.
24. J. Hillston. *A Compositional Approach to Performance Modelling.* Cambridge University Press, 1996.
25. C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall International, London, 1985.
26. Tony Hoare, Bernhard Möller, Georg Struth, and Ian Wehrman. Concurrent Kleene Algebra and its Foundations. 80(6), 2011.
27. S.S. Ishtiaq and P. O'Hearn. BI as an assertion language for mutable data structures. In *Proc. POPL*, 2001. ACM SIGPLAN Notices 36(3), 2001, pages 14–26, `https://doi.org/10.1145/373243.375719`.
28. Julia. `http://julialang.org`. Accessed 22 March 2023.
29. Kim G. Larsen, Ulrik Nyman, and Andrzej Wąsowski. Modal i/o automata for interface and product line theories. In Rocco De Nicola, editor, *Programming Languages and Systems*, pages 64–79, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.
30. Meta. Open-sourcing Facebook Infer. Available at `https://engineering.fb.com/2015/06/11/developer-tools/`. Accessed 22 March 2023.
31. B. Meyer. Applying 'design by contract'. *Computer*, 25(10):40–51, 1992.
32. R. Milner. *A Calculus of Communicating Systems*, volume 92 of *LNCS*. Springer Verlag, 1980.
33. R. Milner. Calculi for synchrony and asynchrony. *Theor. Comput. Sci.*, 25(3):267–310, 1983.
34. R. Milner. *Communication and Concurrency.* Prentice-Hall, 1989.
35. R. Milner. *Communication and Concurrency.* Prentice Hall, New York, 1989.
36. R. Milner. *Communicating and mobile systems: the π-calculus.* Cambridge University Press, 1999.
37. R. Milner. Bigraphs as a model for mobile interaction. In *ICGT 2002, LNCS 2505*, pages 8–13. Springer Verlag, 2002.
38. R. Milner. Bigraphs and their algebra. *Electronic Notes in Theoretical Computer Science*, 209:5–19, 2008.
39. Robin Milner. Bigraphs as a model for mobile interaction (invited paper). In *ICGT 2002, First International Conference on Graph Transformation*, volume 2505 of *LNCS*, pages 8–13. Springer, 2002.
40. Robin Milner. *The Space and Motion of Communicating Agents*. Cambridge University Press, 2009.

41. Bernhard Möller and Tony Hoare. Exploring an interface model for cka. In Ralf Hinze and Janis Voigtländer, editors, *Mathematics of Program Construction*, pages 1–29, Cham, 2015. Springer International Publishing.
42. P.W. O'Hearn and D.J. Pym. The logic of bunched implications. *Bull. Symb. Log.*, 5(2):215–244, 1999.
43. G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Dept., Aarhus University, Aarhus, Denmark, 1981.
44. David Pym. Resource semantics: Logic as a modelling technology. *ACM SIGLOG News*, 6(2):5–41, April 2019.
45. D.J. Pym, P.W. O'Hearn, and H. Yang. Possible Worlds and Resources: The Semantics of BI. *Theor. Comput. Sci.*, 315(1):257–305, 2004.
46. John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proc. LICS 2002*, pages 55–74. IEEE Computer Society, 2002.
47. Colin Stirling. *Modal and Temporal Properties of Processes.* Springer Verlag, 2001.
48. T. Caulfield. SysModels. `https://github.com/tristanc/SysModels`. Accessed 22/03/2023.
49. S. Tripakis, B. Lickly, T. Henzinger, and E. Lee. A theory of synchronous relational interfaces. *ACM Transactions on Programming Languages and Systems*, 33(4):1–41, 2011.
50. Stavros Tripakis. Compositionality in the science of system design. *Proceedings of the IEEE*, 104(5):960–972, 2016.
51. Stavros Tripakis, Ben Lickly, Thomas Henzinger, and Edward Lee. A theory of synchronous relational interfaces. *ACM Transactions on Programming Languages and Systems*, 33(4, Article 14):1–41, 2011.
52. Tuncer Ören and Bernard P. Zeigler and Andreas Tolk (editors). *Body of Knowledge for Modeling and Simulation: A Handbook by the Society for Modeling and Simulation International (Simulation Foundations, Methods and Applications).* Springer, 2023.
53. Johan van Benthem. *Logical Dynamics of Information and Interaction.* Cambridge University Press, 2011.

A A Calculus of Locations, Resources, and Processes

Our starting points are Milner's synchronous calculus of communicating systems, SCCS [33] — perhaps the most basic of process calculi, the collection of which includes also CCS [32], CSP [25], Meije [18], and their derivatives, as well as the π-calculus [36], bigraphs [39] and their derivatives — and the resource semantics of bunched logic [42, 45, 19, 44].

Our presentation here follows Chapter 2 of [12]. We start with the following:

- A monoid of actions, Act, with a composition ab of elements a and b and unit 1;
- The following grammar of process terms, E, where $a \in \mathsf{Act}$ and X denotes a process variable:

$$E ::= 0 \mid a \mid a : E \mid \sum_{i \in I} E_i \mid E \times E \mid X \mid fix_i X.\mathsf{E} \mid (\nu R).E \mid \ldots$$

Most of the cases here, such as 0, action, action prefix, sum, concurrent product, and recursion, will seem quite familiar. The last one, however, which we call hiding, is available because we integrate the notions of resource and process. Its meaning is discussed below; it replaces and generalizes restriction. The expression $fix_i X.\mathsf{E}$, where E is a tuple of processes, means $(fixX.\mathsf{E})_i$, the ith component of the tuple $(fixX.\mathsf{E})$ over an indexing set I. The unsubscripted form $fixX.\mathsf{E}$ stands for the family over i of $fix_i X.\mathsf{E}$ s. As Milner ([33], for example) explains, the case in which the indexing set has just one element is sufficient for finite processes.

Mathematically, the notion of resource — which covers examples such as space, memory, and money — is based on (ordered, partial, commutative) monoids (e.g., the non-negative integers with zero, addition, and less-than-or-equals): each type of resource is based on a basic set of resource elements, resource elements can be combined, and resource elements can be compared. Although, for brevity, we describe the set-up here for resource-process states, R, E, everything we describe here can be set up for location-resource-process pairs, L, R, E, though there are some delicate issues that are beyond the scope of this short paper; see [13, 12, 2].

Formally, we consider pre-ordered, partial commutative monoids of resources, $(\mathbf{R}, \circ, e, \sqsubseteq)$, where $\mathbf{R}$ is the carrier set of resource elements, $\circ$ is a partial monoid composition, with unit e, and $\sqsubseteq$ is a pre-order on $\mathbf{R}$. The basic idea is that resources, R, and processes, E, co-evolve, $R, E \xrightarrow{a} R', E'$, according to the specification of a partial 'modification function', $\mu : (a, R) \mapsto R'$, that determines how an action a evolves R to R' and E to E'.

Modification functions can be seen as the signature of model, specifying the basic evolutions of states via basic actions from which the overall evolution of a model is constructed. In the presence of locations, L, modifications have the form $\mu : (a, L, R) \mapsto (L', R')$, a evolves E to E', R to R', and L to L'.

The base case of the operational semantics, presented in Plotkin's SOS style [43], is given by Action prefix and Concurrent composition, $\times$, exploits the monoid composition, $\circ$, on resources,

$$\frac{}{R, a : E \xrightarrow{a} R', E'} \quad \mu(a, R) = R' \quad \text{Action}$$

$$\frac{R, E \xrightarrow{a} R', E' \quad S, F \xrightarrow{b} S', F'}{R \circ S, E \times F \xrightarrow{ab} R' \circ S', E' \times F'} \quad \text{Concurrent}$$

This (rather general [33, 18]) notion of composition at the level of process does not explain the engineering concept of the composition of models, with its requisite notions of interface and substitution, that we discuss in the sequel.

A modification function is required to satisfy some basic coherence conditions (in certain circumstances, additional structure may be required [2]): for all actions a and b and all resources R and S, and where $\simeq$ is Kleene equality: $\mu(1, R) = R$, where 1 is the unit action, and if $\mu(a, R)$, $\mu(b, S)$, and $R \circ S$ are defined, then $\mu(ab, R \circ S) \simeq \mu(a, R) \circ \mu(b, S)$.

Sums, Recursion, and Hiding are formulated in familiar ways:

$$\frac{R, E_i \xrightarrow{a} R', E'}{R, \sum_{i \in I} E_i \xrightarrow{a} R', E'} \text{Sum} \qquad \frac{R, E_i[fixX.\mathsf{E}/X] \xrightarrow{a} R', E'}{R, fix_i X.\mathsf{E} \xrightarrow{a} R', E'} \text{Recursion}$$

$$\frac{R \circ S, E \xrightarrow{a} R' \circ S', E'}{R, (\nu S)E \xrightarrow{(\nu S)a} R', (\nu S')E'} \text{Hiding}$$

In Recursion, I is an indexing set and E_i is the ith component of a tuple of processes, and substitution is capture-avoiding. In Hiding, the resource S becomes bound to the process E. This construction replaces, and generalizes, the restriction operation of calculi such as SCCS.

The notion of equality between location-resource processes is given by the idea of bisimulation (see [33, 35], for example. Specifically, bisimulation equivalence can be defined as follows: two process states $\mathcal{S}$ and $\mathcal{T}$ are bisimilar, written $\mathcal{S} \approx \mathcal{T}$, if $\approx$ is the largest relation on process states that is closed under the following conditions:

- if $\mathcal{S} \xrightarrow{a} \mathcal{S}'$, then there is an evolution $\mathcal{T} \xrightarrow{a} \mathcal{T}'$ such that $\mathcal{S}' \approx \mathcal{T}'$
- if $\mathcal{T} \xrightarrow{a} \mathcal{T}'$, then there is an evolution $\mathcal{S} \xrightarrow{a} \mathcal{S}'$ such that $\mathcal{S}' \approx \mathcal{T}'$.

For this notion of (strong) bisimulation, two bisimilar processes have the same set of traces/behaviours. Again, this definition of bisimulation can readily be adapted to states L, R, E, though, again, there are some delicate issues that are beyond the scope of this paper; see [13, 12, 2].

B A Modal Logic

Here we follow Chapter 3 of [12]. Process calculi such as SCCS, CCS, and others come along with associated modal logics [23, 35, 47, 53]. Similarly, the calculus sketched here has associated modal logic [13, 12, 2]. The basic logical judgement is of the form $R, E \models \phi$, read as 'relative to the available resources R, the process E has property ϕ'. For our purposes, we work with the following language:

$$\begin{aligned} \phi := \mathrm{p} \mid \neg\phi \mid \top \mid \bot \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \supset \phi \mid \langle a\rangle\phi \mid [a]\phi \\ \mid I \mid \phi * \phi \mid \phi \mathrel{-\!\!*} \phi \mid \langle a\rangle_\nu\phi \mid [a]_\nu\phi \end{aligned}$$

Building on the ideas of the bunched logic BI (e.g., [42, 45, 19, 44]), which is the basis of Separation Logic [27, 46], this logic has, the usual classical connectives, $\top$, $\wedge$, $\rightarrow$, $\bot$, $\vee$ that are defined, in the usual way, by semantic clauses of a satisfaction relation, where $\mathcal{V}$ is an interpretation of propositional letters, beginning

$$R, E \models \mathrm{p} \text{ iff } (R, E) \in \mathcal{V}(\mathrm{p})$$

and proceeding as

$$R, E \models \phi \wedge \psi \text{ iff } R, E \models \phi \qquad R, E \models \psi$$

and so on. Note that an intuitionistic version is also possible. The logic also has a separating conjunction, $*$, with unit I, defined by

$$\begin{aligned} R, E \models \phi * \psi \ \text{ iff there are } S, T \text{ and } F, G \text{ s.t. } S \circ T \sqsubseteq R,\ F \times G \approx E, \\ \text{and } S, F \models \phi \text{ and } T, E \models \psi \end{aligned}$$

where $\approx$ is bisimulation as sketched above, together with a corresponding separating implication, $\mathrel{-\!\!*}$. Note that the truth condition for $*$ — called a 'separating conjunction', since its conjuncts use separate resources — requires the combination of resources from the truth conditions for its component formulae.

The relationship between truth and action is captured by the clauses of the satisfaction relation for the (additive) modalities, given essentially as follows (recall that $R' = \mu(a, R)$):

$$\begin{aligned} R, E \models \langle a\rangle\phi \text{ iff there exists } E' \text{ s.t. } R, E \xrightarrow{a} R', E' \text{ and } R', E' \models \phi \\ R, E \models [a]\phi \text{ iff for all } E' \text{ s.t. } R, E \xrightarrow{a} R', E',\ R', E' \models \phi \end{aligned}$$

Similarly, in addition to the usual quantifiers and modalities, this logic has the separating modalities, $\langle a\rangle_\nu\phi$ and $[a]_\nu\phi$, as developed in [13, 12, 2].

Predicate versions of this logic — of the kind required for this paper — are also available, allowing quantification over both actions and terms. The details require some care, and will reported elsewhere, but the basic idea is that quantification over action variables taking values in Act and term variables taking values in a specified domain $\mathcal{D}$, as part in general of a model $\mathcal{M}$ defined in a standard way, is understand as for states S as

$$\begin{array}{l}
\mathcal{S} \models_{\mathcal{M},\rho} \exists \alpha.\phi \text{ iff there exists } a \in \mathsf{Act} \text{ such that } \mathcal{S} \models_{\mathcal{M},\rho[\alpha:=a]} \phi \\
\mathcal{S} \models_{\mathcal{M},\rho} \forall \alpha.\phi \text{ iff for all } a \in \mathsf{Act},\ \mathcal{S} \models_{\mathcal{M},\rho[\alpha:=a]} \phi \\
\mathcal{S} \models_{\mathcal{M},\rho} \exists z.\phi \text{ iff there exists } d \in \mathcal{D} \text{ such that } \mathcal{S} \models_{\mathcal{M},\rho[z:=d]} \phi \\
\mathcal{S} \models_{\mathcal{M},\rho} \forall z.\phi \text{ iff for all } d \in \mathcal{D},\ \mathcal{S} \models_{\mathcal{M},\rho[z:=d]} \phi
\end{array}$$

where ρ assigns values in Act and $\mathcal{D}$ and $\models_{\mathcal{M},\rho}$ denotes satisfaction in a model $\mathcal{M}$ in the usual way. We elide the detail of the connection to the implemented models discussed in this paper.

The basic connection between the process calculus and the logic is given by a form of van Benthem-Hennessy-Milner theorem that relates process equivalence, as given by bisimulation, and logical equivalence (e.g., [23, 35, 47, 13, 53, 12, 2]). Essentially — though some care must be taken [12, 2], either restricting the logic or employing a slightly different semantics — for image-finite processes E and F and any R,

$$R, E \approx R, F \quad \text{iff} \quad R, E \equiv_{\text{MBI}} R, F$$

Specifically, this holds for the basic monoidal theory of resources, as described above, only for some (large) fragments of the logic [13, 12], but, with a slightly different theory of resource — essentially, using two monoidal operations — as described in [2], it holds for the full logic with states R, E and S, F (and hence L, R, E and M, S, F). This generalization is slightly mis-stated in [9].

Logics based on the language of MBI have proved valuable in program analysis — see the Infer tool [30] — partly by virtue of their deployment of local reasoning, based on the connective $*$.

C Definitions of Interfaces in the Literature

In this appendix we briefly discuss how interface concepts from the systems and modelling literature relate to our ideas on interface presented in this paper.

Interface automata The first formal interface theory was proposed by de Alfaro and Henzinger in [1]. There, an interface theory comprises an interface algebra together with a component algebra to differentiate the interface specifications from component implementations. An interface is represented by an input/output automaton, which is an automaton whose transitions are labelled with input or output actions. The main requirement in this setting is that connection is properly accomplished in the sense that for every output one interacting component produces, the other component is ready to receive it.

This approach corresponds to the case when the models have only basic interfaces. Our interface models extend this type of interaction, allowing for some possible transformations of the model outputs before being received by another model.

Modal I/O Automata for Interface A modal transition system (MTS) [29] associates with each state transition, a classification/modality — may or must — expressing transition behaviours that (i) necessarily occur (must modality), (ii) possibly occur (may modality), and (iii) not possibly occur (absence of a transition). Modal I/O Automata for Interface is an approach that uses combinations of Interface Automata (IA) and Modal Transition Systems (MTS), through the labelling of may and must modalities with input/output features.

The modal I/O Automaton model can be thought of as a particular case of our interface model in which we express the may and must modalities using appropriate logical formulae.

Relational Interfaces The theory of relational interfaces [49] specifies interfaces as relations between inputs and outputs. This interface theory includes explicit notions of environments and the compatibility of the interface with the environment.

Relational interfaces can be thought of as a particular case of our interface models in which there is no internal structure, and the logical formulae model the relationships between inputs and outputs.

Contract theories Contract theories [31, 4, 5] are specializations of interface models in the sense that they obey to the concept of separation of concerns. They separate the assumption specification from the guarantees specification. The first use of the concept of contract [31] specifies preconditions and postconditions as state predicates for the methods of a class, and invariants for the class itself. The precondition expresses requirements that any operation has to satisfy if it is to be correct; the postcondition expresses properties that are ensured after the execution of the operation. In [4], the notion of a contract for a component is built as a pair of assertions, which express its assumptions and promises. The assertions talk about the entire component behaviour. Then, a contract is an assertion on the component behaviors (the promise) subject to certain assumptions.

The contract model can be seen as a special example of our interface model with no internal structure where the contract (assume-guarantee) assertions are special logical formulae about the ecosystem components.

Interfaces in Concurrent Kleene Algebra Concurrent Kleene algebra (CKA) [26] has been defined as an algebraic framework to reason about concurrent programs. CKA basic ingredients are events, which are occurrences of certain primitive actions. A set of events is called a trace and represents a program or a specification. A trace defines a graph with events as nodes and arrows corresponding to dependences like causal or temporal succession and the like. The basic idea of the interface model [41] is to consider subgraphs, called graphlets, of that overall graph as events of their own. The sets of incoming and outgoing arrows of a graphlet constitute its input/output interface.

In this case, the interface model defined as a graphlet can be directly related to our model of basic interface. Note that, in our approach, we consider not only the incoming and outgoing arrows, but also the internal structure of the subgraph that models the basic interface.

Bigraphs Bigraphs [40, 37, 38] represent a formalism designed for modelling agents, their spatial arrangement, and their inter-connections. The formalism for describing a bigraph can be done either as a graph or an algebraic form. A bigraph is modelled as the superposition of a graph (the link graph) and a set of trees (the place graph). In the graphical description, the nodes encode agents or other entities. The spatial configuration is described by node nesting. The vertices have associated controls that form a signature. As well, the nodes have an arity (i.e., the number of ports for a given node to which link-graph edges may connect). A bigraph may have associated with it one or more regions (or roots) which describe adjacent parts of the system. In the bigraph structure, we have also *sites*. Nodes, sites and roots are the places of a bigraph.

Formally, a bigraph is defined as:

$$F = (\mathbf{V}, \mathbf{E}, ctrl, prnt, link) : \langle k, X\rangle \to \langle m, Y\rangle$$

thought of as a superposition of a place graph $F_P = (\mathbf{V}, ctrl, prnt) : k \to m$ and a link graph $F_L = (\mathbf{V}, \mathbf{E}, ctrl, link) : X \to Y$

A bigraph has two faces: an outer and an inner interface. Each of these interfaces is a pair containing a face for the place graph and a face for the link graph. For a place graph, a face is a natural number. In this framework we treat a natural number as the set of numbers that are strictly smaller; that is, $n = \{0, 1, \ldots, n-1\}$.

For a link graph, a face is set of labels — that is, those attached to the unconnected edges. A bigraph can have inner names and outer names, which define the bigraph interfaces.

A bigraph is able to interact with its external environment through its interfaces. When the interfaces of two bigraphs are matching, we can define their composition.

Because a bigraph is built as the superposition of two graphs (link graph and place graph), the corresponding interfaces can also be thought of as a superposition of the interfaces associated with the component graphs. Even though the component graph interfaces are different objects, the essence of their definition is the same: input/output variables. Therefore, we can link these interfaces with our basic interfaces.