The Principle of Comprehensive Modelling Applied to Data and Behaviour

Ashley McNeile
Metamaxim Ltd.
48 Brunswick Gardens
London, U.K.
ashley.mcneile@metamaxim.com

ABSTRACT

In their paper "Trace Semantics are Fully Abstract" Nain and Vardi propose that denotational semantics of behavioural modelling formalisms should adhere to the "Principle of Comprehensive Modelling" which requires models are exhaustive of all possible circumstances. However their exploration of this principle is made only in the context of machines that use sets of single symbols to define their state and alphabet. We extend the application of the principle to the denotational semantics of machines that use data structures, rather than single symbols, for the definition of state and alphabet. We show that, in the context of modelling the interaction between data and behaviour, this extension enables a unification of what are normally considered distinct styles of behavioural composition.

Categories and Subject Descriptors

F.1.1 [**Computation by Abstract Devices**]: Models of Computation—*automata*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Denotational semantics*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Process models*

General Terms

Semantics, Computation, Composition

Keywords

Process models, Denotational semantics, Comprehensive modelling, Composition, Interaction

1. INTRODUCTION

The usual definition of a *deterministic finite automaton* (DFA) is as follows:

A deterministic finite automaton is a 5-tuple:[1]

$$\langle \mathcal{A}, \mathcal{S}, \mathcal{T}, s_0, \mathcal{F} \rangle$$

consisting of:

- a finite set of input symbols called the alphabet ($\mathcal{A}$)
- a finite set of states ($\mathcal{S}$)
- a transition function ($\mathcal{T} : \mathcal{S} \times \mathcal{A} \rightarrow \mathcal{S}$)
- a start state ($s_0 \in \mathcal{S}$)
- a set of accept states ($\mathcal{F} \subseteq \mathcal{S}$)

This definition can be described as "comprehensive" in that the behaviour of the DFA, embodied in the transition function $\mathcal{T}$ is defined for every member of $\mathcal{S} \times \mathcal{A}$; so whatever the state of the DFA $\mathcal{T}$ defines a new state (which could be the same as the old one) for every member of $\mathcal{A}$.

Not all behavioural models are comprehensive in this sense. For instance, the UML Superstructure [5] pages 546, 547 gives semantics of a protocol state machine:

Transitions of protocol state machines have the following information: a pre-condition (guard), on trigger, and a post-condition. Every protocol transition is associated to zero or one operation (referred BehavioralFeature) that belongs to the context classifier of the protocol state machine.
...
The interpretation of the reception of an event in an unexpected situation (current state, state invariant, and pre-condition) is a semantic variation point: the event can be ignored, rejected, or deferred; an exception can be raised; or the application can stop on an error. It corresponds semantically to a pre-condition violation, for which no predefined behaviour is defined in UML.

This statement says that the semantics of UML state machines is **not** comprehensive, as otherwise there could be no such thing as an "unexpected situation".

2. COMPREHENSIVE MODELLING

Nain and Vardi have suggested a *Principle of Comprehensive Modelling* [7], whereby the semantics of behavioural models should always be comprehensive. They argue that:

[1]Based on the Wikipedia entry for DFA: `http://en.wikipedia.org/wiki/Deterministic_finite_automaton`

The Principle of Comprehensive Modeling requires a process description to model all relevant aspects of process behaviour. From this point of view, certain process-algebraic formalisms are underspecified, since they leave important behavioral aspects unspecified. For example, if the distinction between normal termination and deadlocked termination is relevant to the application, then this distinction ought to be modeled explicitly.

The UML protocol state machine semantics is an example of exactly this kind of under specification. If their principle were followed there would be no need for a "semantic variation point" as the treatment of all situations would be covered in the semantics. In their papers, Nain and Vardi argue that the Principle of Comprehensive Modelling contributes to ensuring that trace semantics of behavioural models are *fully abstract.*

Nain and Vardi illustrate their ideas using machines (abstract transducers) whose actions and states are defined using sets of single symbols, in a style common in mainstream automata theory and process algebras. In this paper, we assume that the case for adopting the Principle of Comprehensive Modelling is made (we are not repeating the arguments made by Nain and Vardi) and consider what it means when we apply it to the definition of the semantics of behavioural models whose actions and state are arbitrary data structures, so that models represent the interplay of data and behaviour. In this context we explore, in particular, the implications of adopting this principle on the interpretation of compositions, where one process sends data which another process receives.

3. A SIMPLE MODELLING NOTATION

For the purposes of examining the meaning and implication of the Principle of Comprehensive Modeling we will consider its application to a form of state machine which we call an Independent Protocol Machine (IPM).[2] This machine is similar in concept to the DFA considered in Section 1 except that the actions and states are not single symbols but data structures. We first construct a trace-based denotational semantics for these machines, and start with a formalization of data.

3.1 Formalization of Data

The basic notion we use for formalizing data is that of a *value expression.* A value expression associates a value with a symbol and takes the form `(symbol=value)`, for example `(s1=6)` and `(s2="frog")`. The domains of *symbols* (for the left hand side of values expressions) and *values* (for the right hand side of values expressions) are disjoint and this means that a value cannot reference a symbol or vice versa. So `(s1=s2)` is not legal. Two value expressions are equal iff both the symbol and value parts of the two expressions are equal.

If w is a value expression w^{symb} gives the symbol part and w^{val} gives the value part. So if w is the value expression `(s1=6)` then $w^{symb} =$ `s1` and $w^{val} =$ `6`.

[2]This is a simplified form of Protocol Machine, described, for instance, by the author and Simons [1].

We denote:

- The set of all possible symbols by $\mathcal{Y}$.
- The set of all possible values by $\mathcal{V}$.
- The set of all possible value expressions by $\mathcal{E}$.

Note that $\mathcal{E} = \mathcal{Y} \times \mathcal{V}$ and represents every expressible value expression, whether it has defined meaning or interpretation or not. We assume that $\mathcal{Y}$ and $\mathcal{V}$ are finite, and this means that $\mathcal{E}$ is also finite. However we take the number of symbols and the number of values to be very large, so large that there is no use or application of the theory that can come anywhere near exhausting the supply.

A set of value expressions is *consistent* if it contains at most one entry for a given symbol. Thus the set of value expressions:

```
{(s1=5),(s2=3),(s1=4)}
```

is **not** consistent because it contains two value expressions for the symbol `s1`. We use the function *con* with signature:

$$\mathit{con} :: \mathit{power}(\mathcal{E}) \rightarrow \mathit{boolean}$$

to indicate that a set W of value expressions is consistent, as follows:

$$\begin{aligned} &\mathit{con}(W) \;\Leftrightarrow \\ &\quad w1, w2 \in W \wedge w1^{val} \neq w2^{val} \Rightarrow w1^{symb} \neq w2^{symb} \end{aligned} \quad (1)$$

3.2 Formalization of Behaviour

Here we develop a trace-based semantics for an IPM. We use the concept of a *completion*, being similar to the familiar idea of a trace, but describing the data of the machine. A *completion* is described as a sequence of *steps.* A step describes the behavior of a machine for a single action. Each step is a triple $s = \langle a, s, d \rangle$ where:

- $a \in \mathit{power}(\mathcal{E})$ is a set of value expressions that is the *action* of the step.
- $s \in \mathit{power}(\mathcal{E})$ is a set of value expressions that is the *state* of the machine at the **end** of the step.
- $d \in \{\mathit{allow}, \mathit{refuse}, \mathit{crash}\}$ is the *decision* that P takes on whether to *allow* or *refuse* the *action* of the step.

A decision of *allow* means that the machine moves to the new state defined by s. A decision of *refuse* means that the machine has no defined response to the action and does not change state. A decision of *crash* means that machine undergoes an irrecoverable failure. There is an important conceptual distinction between *refuse* and *crash*:

- Reaching a *refuse* does not mean that execution of a machine is finished.[3] If a machine refuses an action a it remains in the state that pertained before the step for a and is free to engage in another action, reacting exactly as though the a had never occurred.

[3]In this respect, the term "completion" is perhaps misleading.

- Reaching a *crash* marks the end of execution of the machine. Trying again after a *crash* is meaningless as the machine is in an incoherent state and has no specified behaviour.

A *completion* of P is a sequence of steps that provides a partial description of P's behaviour by describing a possible execution scenario. A completion has either:

- an infinite number of steps, all with a decision of *allow*; or

- a finite number, one or more, of steps each with a decision of *allow* followed by a single step with a decision of *refuse* or *crash*.

The behaviour of a machine is defined as a set of completions. Note that because a machine can continue to process further actions after a *refuse*, the set of completions for a machine is not the same as the set of execution scenarios. However, the set of completions of a machine serve to specify all possible execution scenarios exhaustively. We now apply the Principle of Comprehensive Modelling (PCM) to determine the extent of this set.

3.3 Application of the PCM

We can frame the question of how to comply with the PCM as follows. Suppose that t is an incomplete completion for a machine P, in other words a finite sequence of steps of P all with a decision of *allow* representing a possible partial execution scenario for P. We need to identify the set $next(t)$ of next steps such that $\{t^\frown s \mid s \in next(t)\}$ (where $t^\frown s$ means t extended by the step s) gives a comprehensive definition of what can happen in the next in PP following t. To answer this we have to consider the universe of data possibilities that $next(t)$ has to accommodate.

We suppose that we have complete knowledge of the symbols that the machine P can use as $\mathcal{A} \cup \mathcal{S}$ where:

- $\mathcal{A} \subseteq \mathcal{Y}$ is the set that can be used in actions

- $\mathcal{S} \subseteq \mathcal{Y}$ is the set that can be used in states

with $\mathcal{A} \cap \mathcal{S} = \emptyset$. Any step $\langle a, s, d\rangle$ in P has $symb(a) \subseteq \mathcal{A}$ and $symb(s) \subseteq \mathcal{S}$. We use this to define the data universe $\mathbb{U}$ of P as:

$$\mathbb{U} = \{\ W \mid W \in power(\mathcal{E}) \ \wedge \\ symb(W) = (\mathcal{A} \cup \mathcal{S}) \ \wedge \ con(W)\ \} \tag{2}$$

where $symb(W) = \{\ y \mid w \in W \wedge w^{symb} = y\ \}$. To satisfy the PCM we then require that $next(t)$ meets:

$$\forall\ u \in \mathbb{U} \ \ \exists \langle a, s, d\rangle \ \in next(t) \text{ with } \ (a \cup s) \subseteq u \tag{3}$$

This requires that, for every element u of the data universe $\mathbb{U}$, P has at least one step with a data image (the union of the action and the state parts of the step) that matches (is contained in) u. Note that (3) does require that P allows any action in any state as the decision part of a step can be *refuse* or *crash*. Rather, it requires that the behaviour of P is always *defined*.

The denotational semantics of *completions* bears a close resemblance to the *Stable Failures* denotational model used in CSP, as described by Roscoe et al. [2]. A *failure* is a trace (of allowed actions) along with a set that contains all the actions that could then be refused. This enables distinction to be made between cases where, because of non-determinism, a simple traces model does not suffice. The classic example is shown in Figure 1, where two processes have the same traces but different behaviour. The traces model for the two processes is the same, but both the failures and the completions models (using steps with just *action* and *decision*) distinguish the two.

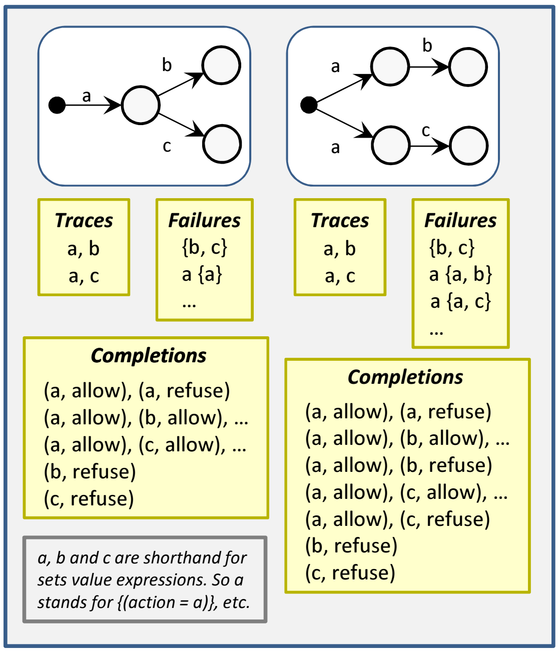

Figure 1: Traces, Failures and Completions

The right hand example in Figure 1 is non-deterministic and, as this example suggests, using failures or completions effectively addresses non-determinism. However if we wish to restrict machines to be deterministic we must require that, in addition to (3):

$$\forall\ u \in \mathbb{U} \text{ and } a \text{ with } symb(a) \subseteq \mathcal{A} : \\ |\ \{\ \langle a, s, d\rangle \ \in next(t) \mid a \subseteq u \ \ \wedge \ \ d = allow\ \}\ | < 2 \tag{4}$$

which states that there is at most one element of $next(t)$ which is allowed for any given action, all others either being refused or resulting in a crash.

4. VALID DATA

The formalization of the universe given in (2) allows the symbols of $\mathbb{U}$ to take any value allowed in $\mathcal{E}$. However we must recognise that the symbols in $\mathcal{A} \cup \mathcal{S}$ will have domain semantics which restrict the values that it can meaningfully take. For example the symbol *balance of account* can only meaningfully take numeric values, and the symbol *date of birth* can only meaningfully take date values. Based on this we suppose that every symbol has an associated data type and we identify $\mathbb{V} \subseteq \mathbb{U}$ as the *valid universe* for P, where $\mathbb{V}$ represents valid data, by conforming to the following two conditions. If $V \in \mathbb{V}$ and $v \in V$ then:

1. The value expression v has a value that conforms to the data type (numeric, date, boolean, etc.) of its symbol.
2. If v is a derived symbol (one whose value is computed from other data) then its value is correctly computed.

For the purpose of the second of these conditions we identify a subset $\mathcal{D} \subseteq (\mathcal{A} \cup \mathcal{S})$ of the symbols of P's universe as set of derived symbols. For each symbol $d \in \mathcal{D}$ we have a set of symbols $\mathit{basis}(d) \subseteq (\mathcal{S} \backslash \mathcal{D})$ which is the set of symbols used to calculate (derive) the value of d. An analogy is a *spreadsheet*, where some cells contain values and some contain formulas. The latter are the derived cells. The formula of a derived cell, c, can reference other cells that are also derived, but the recursive closure of such references must be a set of cells that contain values (which are not derived), and this set of cells is the *basis* of c.

As a matter of convenience, and without loss of generality, we assume that the basis set of a derived symbol does not contain other derived symbols. We also assume that a basis set of a derived symbol does not contain action symbols, which also entails no loss of generality as we can assume that any non-derived action symbol is mirrored by a symbol of the state of the machine, and then use this mirrored symbol in the basis instead.

To be well-formed the valid universe has the property that:

$$\begin{aligned} &\forall\, y \in \mathcal{D} \text{ and } W1, W2 \in \mathbb{V} \text{ with} \\ &\quad (\{y\} \cup \mathit{basis}(y)) \subseteq \mathit{symb}(W1 \cap W2) : \\ &\quad \mathit{restr}(W1, \mathit{basis}(y)) = \mathit{restr}(W2, \mathit{basis}(y)) \Rightarrow \\ &\quad \mathit{restr}(W1, \{y\}) = \mathit{restr}(W2, \{y\}) \end{aligned} \tag{5}$$

where $\mathit{restr}(W, \mathcal{Z}) = \{\, w \in W \mid w^{symb} \in \mathcal{Z} \,\}$. This requires that two members of the valid universe that give the same values to the basis set of a derived symbol must also give the same value to the derived symbol itself. This is clearly required, otherwise derivation is non-deterministic.

In order to conform to the PCM the behaviour of a machine must be defined in terms of the full universe, $\mathbb{U}$, as we have done in (3). However, we need to consider within this comprehensive definition of behaviour what happens in the event of departure from the subset $\mathbb{V}$ of $\mathbb{U}$ that constitutes valid data. Here we take it that departure from the valid universe cannot be successfully tolerated, so if t is a partial completion and $\langle a, s, d\rangle \in \mathit{next}(t)$ then:

$$(a \cup s) \not\Subset \mathbb{V} \Rightarrow d \neq \mathit{allow} \tag{6}$$

where $W \not\Subset \mathbb{V} \Leftrightarrow \forall\, V \in \mathbb{V},\ W \nsubseteq V$. The condition (6) says that any step whose data image does not match any element of the valid universe must be refused or it will result in the machine crashing. A well designed machine would ensure that steps that would result in departure from $\mathbb{V}$ would have a decision of *refuse*, so that the machine would not crash. By refusing the action, the machine remains alive and able to take another step. Not all software is well designed, so the PCM requires that *crash* is a modelled possibility.

Modelling all the possibilities, including those that cause the machine to crash, leaves no room for the possibility of the kind of "unexpected situation" mentioned in the UML specification quoted at the end of Section 1.

5. COMMUNICATION

Derived symbols in the data image of a step in the behaviour of a machine represents *computation*, in the following sense. Suppose that the state of a machine contains an integer valued symbol and that $y \in \mathcal{S} \cap \mathcal{D}$ is a boolean, also in the state of the machine, which says whether the integer is prime or not. The symbol y is an abstraction of the integer in the sense that, if the integer value were somehow lost, it would not be possible to re-establish it from y; but if y were lost it could be re-established from the integer. This is only possible because y is a derived symbol and can be re-established by computation. The implication of by (6) is that if the machine is not to crash, it must compute derived values to ensure that its data image remains inside $\mathbb{V}$. In this example, the machine that owns y as part of its state must correctly compute whether or not the integer that is the basis for y is prime.

5.1 Derived Symbols in Actions

In the case where a derived symbol belongs to an action, so $y \in \mathcal{A} \cap \mathcal{D}$, we identify the symbol with an *output* of the machine, this being the natural interpretation of a computed value in an action. So in any step $\langle a, s, d\rangle$ we have, in general, a mixture of input and output symbols in its action:

$$\begin{aligned} &\text{Input: } \mathit{restr}(a, \mathcal{A} \backslash \mathcal{D}) \\ &\text{Output: } \mathit{restr}(a, \mathcal{D}) \end{aligned} \tag{7}$$

where the output symbols represent the machine's "response" to the input, in the same manner as a function returns a value based on its input parameters (although a protocol machine cannot really be equated with a function, as a protocol machine has state). With this identification we can look at the modelling of machines that communicate.

5.2 Traditional Semantics of Interaction

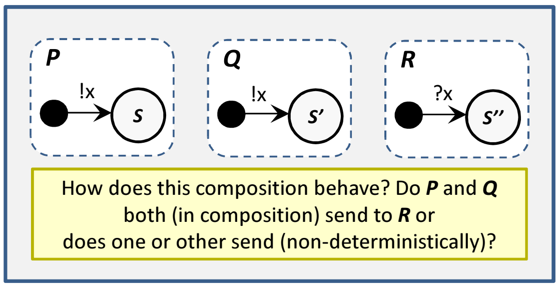
P
!x
S
Q
!x
S'
R
?x
S''
How does this composition behave? Do P and Q both (in composition) send to R or does one or other send (non-deterministically)?

Figure 2: Communicating processes

Consider the composition of processes shown in Figure 2. Following the usual convention, we use ! to denote output (sending) and ? to denote input (receiving). Superficially the situation admits two different interpretations:

- Both P and Q can interact R but only one may do so. So either P interacts with R, in which case P advances to state S and R to state S'' but Q remains at its initial state, $\bullet$; or Q interacts with R, in which case Q advances to state S' and R to state S'' but P remains at its initial state. The system has to make a non-deterministic choice between these two possibilities.
- We view P and Q as in composition on their common action $!x$ and so both, co-operatively, engage in this

event in order to interact with the $?x$ event in R. In this interpretation all three processes advance, to S, S' and S'' respectively, and there is no non-deterministic choice involved.

Traditionally this choice is positioned as a choice about the semantics of composition. In his *Calculus of Communicating Processes* (CCS [6]) Milner takes the first choice; however in *Communicating Sequential Processes* (CSP [4]) Hoare takes the second. (In his paper *A Comparative Introduction to CSP, CCS and LOTOS* [3] Fidge gives a good summary of differences between the semantics of composition in CCS and CSP.) However, armed with the exploration of the treatment of data given in protocol machines given above, we can obtain a quite different view of this choice.

5.3 The Implication of Data Universes

In Section 3.3 we argued that, in order to comply with the PCM and furnish a complete definition of the behaviour of a machine, it is necessary to identify a universe of all the data possibilities that the machine must accommodate. If we have two machines in composition, we have to ask: Are they using the same universe or different universes? We show that this question has a direct bearing on the semantics of the composition.

We argued in Section 5.1 that derived symbols in an action represent outputs and that, according to (5) the relationship between the values of a derived symbol and its base symbols are captured by well-formedness rules on a universe. Suppose we have two distinct universes, $\mathbb{U}$ and $\mathbb{U}'$ with at least some symbols in common, so $(\mathcal{A} \cup \mathcal{S}) \cap (\mathcal{A}' \cup \mathcal{S}') \neq \varnothing$. Suppose further that each universe has its own subset of valid data, $\mathbb{V}$ and $\mathbb{V}'$ respectively. If they are to be used together the two universes must have disjoint sets of the derived symbols:

$$\mathcal{D} \cap \mathcal{D}' = \varnothing \tag{8}$$

otherwise they could define different and conflicting derivations of the same symbol, resulting in an incoherent system.

If we interpret Figure 2 as a picture of protocol machines we assume that the symbols involved in the actions labelled $!x$ in P and Q are derived, as they are output. Suppose that P and Q are defined using different universes $\mathbb{U}$ and $\mathbb{U}'$, and suppose that the steps in P and Q responsible for the output are $\langle a, s, d\rangle$ and $\langle a', s', d'\rangle$ respectively. By (8) the actions in P and Q cannot share any symbols, so $a \cap a' = \varnothing$. While it is possible that the universes used by P and Q are clones, so that a and a' are isomorphic, the symbol sets they use are nevertheless disjoint. This means that P and Q do not co-operate in the production of the output; and, assuming that R has possible steps defined for both a and a', either one could interact with R. This corresponds to the CCS semantics for interaction.

Now we suppose that P and Q are defined using the same universe. In this case, (8) does not apply and the action parts a and a' of the output steps in P and Q may share symbols, and could even be identical. Assuming some shared symbols between them, and that $con(a \cup a')$ so that there is an element u of their shared universe for which both obey (3), both P and Q will engage simultaneously in production of the output. This corresponds to the CSP semantics for interaction.

6. CONCLUSION

In this paper we have used two ideas:

- the Principle of Comprehensive Modelling, as proposed by Nain and Vardi, and
- modelling data in behavioural models, to enable the representation of the interaction between data and behaviour.

We have demonstrated that, used together, these ideas enable accommodation of two styles of composition, that used in CCS and that used in CSP, within a single framework. What is conventionally treated as a choice of semantics, and therefore globally hard-wired into the language, becomes a local distinction based on whether the composed processes share a single data universe or use different universes. This seems a superior treatment of the dichotomy, as it does not prefer one choice of semantics over the other.

References

[1] A. McNeile and N. Simons. Protocol Modelling: A Modelling Approach that supports Reusable Behavioural Abstractions. *Journal of Software and System Modeling*, 5(1):91–107, 2006.

[2] A. Roscoe, C. Hoare, and R. Bird. *The Theory and Practice of Concurrency.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.

[3] C. Fidge. A Comparative Introduction to CSP, CCS and LOTOS. Technical report, The University of Queensland, Queensland 4072, Australia, 1994.

[4] C. Hoare. *Communicating Sequential Processes.* Prentice-Hall International, 1985.

[5] Object Management Group. Unified Modeling Language, Superstructure: Version 2.4.1. *OMG Document Number: formal/2011-08-06*, 2011.

[6] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science.* Springer, 1980.

[7] S. Nain and M. Vardi. Trace Semantics is Fully Abstract. In *Proceedings of the 2009 24th Annual IEEE Symposium on Logic In Computer Science*, LICS '09, 2009.