Complexity as a Basis for Comparing Semantic Models of Concurrency

Alan Mycroft[1], Pierpaolo Degano[2] and Corrado Priami[2]

[1] Computer Laboratory, Cambridge University,
New Museums Site, Pembroke Street, Cambridge CB2 3QG, United Kingdom
`Alan.Mycroft@cl.cam.ac.uk`
[2] Dipartimento di Informatica, Università di Pisa
Corso Italia 40, I-56100 Pisa, Italy
`{degano,priami}@di.unipi.it`

Abstract. We investigate some aspects of interexpressiveness of languages and their (denotational) semantic models by viewing semantic functions from a complexity-theory viewpoint. We classify semantic functions as *polynomial* or *finite* if a language term of size n produces a meta-language object respectively polynomially bounded in n or finite. Languages involving concurrency manifest most interest which we associate to the fact that their semantic models in general lack λ-style abstraction. The paper provides a quantifiable reason why labelled event structures form a more reasonable model for the choice and concurrency operators of CCS than do synchronisation trees. Similarly we show the representation of conflict by places within (at least occurrence forms of) Petri-nets is exponentially larger than the relational representation within corresponding event structures. An application is a criterion for selection of semantic models for real-world algorithmic purposes; for example, 'model checking' algorithms which use an exponentially larger semantic representation of programs are unlikely to be efficient.

1 Introduction

The idea of a denotational semantics has several components. Firstly we have an *(object)-language* whose (freely constructed) terms are to be given meaning. We also have a *semantic function* which homomorphically maps object terms into terms of a *meta-language*. Finally, and importantly, we have an equivalence relation on meta-language terms specifying when they are equal. For the typed λ-calculus with constants as a meta-language this is typically $\alpha\beta\eta\delta$-equality. Here we view object- and meta-language terms as *data-structures with a given representation*—two different representations for meta-language terms would count as two separate semantic models (see section 2.2 for more discussion). Also note our emphasis on the separateness of translation into meta-language terms and their equality—this is done in order to study the size of meta-language terms which obviously, in general, is not preserved by replacing a term with an equal one.

For classical programming languages (Pascal for example) the use of typed λ-calculus as a meta-language for their denotational semantics à la Scott is widespread and undisputed. One debates the meaning of a given programming language feature (e.g. parameter passing mechanism) *within* the meta-language. By contrast, there is a plethora of models for concurrent systems and it seems at times that each feature requires a new meta-language to describe it (witness *inter alia* the various forms of event structures).

In spite of the huge amount of theoretical work on semantics for concurrent systems, there is a lack of studies to show how practically useful concepts and theories are. Unfortunately, the choice among different approaches is quite often based on a matter of taste, rather than on comparable and measurable grounds. This is a major drawback because it allows the proliferation of proposals without a measure of their relative merits. This paper amounts to one proposal for this measure.

1.1 Languages

We will consider several languages and models in this paper. Following the definition style of process algebras, our general language $\mathcal{L}$ has *ports* ranged over by α, *co-ports* ranged over by $\overline{\alpha}$ (the sets of ports and co-ports yield the set of *actions*), *(process) names* ranged over by P to yield behaviours Beh, ranged over by b:

$$b ::= nil \mid \alpha.b \mid \overline{\alpha}.b \mid \tau.b \mid b + b' \mid b \mid b' \mid b; b' \mid b[\phi] \mid b\backslash\alpha \mid P \mid nonrec\, P.b \mid rec\, P.b.$$

The nil operator denotes a process that can perform no action; $\alpha.b$ ($\overline{\alpha}.b$ or $\tau.b$) is a process that fires α ($\overline{\alpha}$ or τ) and then behaves like b; $b + b'$ is a process that non-deterministically behaves like b or like b'; $b \mid b'$ is a process that can perform independently both b and b'; $b; b'$ is a process that performs b and when b stops, b' is executed; ϕ is a relabelling function preserving τ and $\bar{\cdot}$, rec allows recursive process definitions and $nonrec$ allows simple non-recursive definitions—see below. In the sequel, we assume processes guarded and closed. In examples, we will abbreviate processes like $\alpha.nil$ to α. Sublanguages of $\mathcal{L}$ will be denoted $\mathcal{L}[op_1, \ldots, op_n]$, these always include process names P (if rec or $nonrec$ present), ϕ and nil. For example, $\mathcal{L}[\alpha, \overline{\alpha}, +, |, \phi, \backslash, rec]$ coincides with the CCS [19], and $\mathcal{L}[\alpha, +, rec]$ refers to the sublanguage given by

$$b ::= nil \mid \alpha.b \mid b + b' \mid P \mid rec\, P.b$$

which is very close to the idea of classical regular expressions. The above precision (for example separating rec and $nonrec$) is important because we argue about complexity results which alter greatly with changes to semantic power. However, in examples, we will feel free to use Milner's original notation of

$$\begin{array}{l} P_1 \Leftarrow b_1 \\ \ldots \\ P_n \Leftarrow b_n \\ in\ b_0 \end{array}$$

instead of rec and $nonrec$.

Languages (which are later defined to be free algebras) have a simple notion of *size*, for example the number of constructors.

1.2 Semantic models

Here we take a semantic model to be a meta-language (possibly including infinite terms, see below) together with an equivalence relation stating when two terms are equivalent (for example the λ-calculus equality includes β-equivalence). Our semantic models will be taken to include (besides the typed λ-calculus with constants and recursive type definitions) *labelled transition systems* [17]. These are triples $TS = \langle \mathcal{T}, A, \rightarrow \rangle$ where $\mathcal{T}$ denotes the states of the transition system, i.e. programs ranged over by $t, t', t_1, \ldots$, A is the labelling alphabet ranged over by $\alpha, \beta, \ldots$ and $\rightarrow \subseteq \mathcal{T} \times A \times \mathcal{T}$ is the transition relation labelled on A. The transition relation specifies the possible changes of states when a program is run. We will write $t \xrightarrow{\alpha} t'$ for $(t, \alpha, t') \in \rightarrow$. Roughly, a transition system is a graph whose nodes are programs and whose arcs are labelled by actions.

Note that any sub-language of $\mathcal{L}$ which does not contain the *rec* operator generates acyclic labelled transition systems under the natural semantic functions. We will use the name *synchronisation DAG* to refer this special kind of transition systems.

Another special case of transition systems are *trees*. They are generated under the assumption that any transition leads to a new node (possibly a copy of one already existing). This is the intuitive idea of unfolding of transition systems. We will write trees as summations with *NIL* for the empty tree. Given a term t, the tree of t is $[\![t]\!] = \sum_{i \in I} \alpha_i.[\![t_i]\!]$ for any $t \xrightarrow{\alpha_i} t_i$ in the transition system of t. Examples of trees used to represent the behaviour of processes are *synchronisation trees* [19], *acceptance trees* [16], *causal trees* [9], *proved trees* [10].

A variant of transition systems are the *compact transition systems* [11]. They are transition systems smaller than the classical ones that still preserve causal relation (but also independency and conflict) between events due to their richer labelling of transitions. Indeed, they are a compact representation of proved transition systems [4] whose transitions are labelled by encodings of their proofs. These compact representations are obtained by exploiting independency and conflict relations between the transitions exiting from the same node. In particular, only one transition (in absence of autoconcurrency) is kept among the mutually concurrent ones.

Also the so-called *algebras of transitions* [12] yield transition graphs. A unique name is associated to each transition (its proof) and composition operators are defined on them according to the constructs of the language which form the algebra of states. As the synchronisation mechanism is completely free (as the treatment of restriction) special transitions labelled *error* must be introduced (e.g. for the (non-existent) synchronisation between α and β in $\alpha \mid \beta$). This implies that more transitions than necessary have to be generated and then, observing the labelling, have to be erased. Moreover, as the name of the transitions is recorded in the states, there is also an explosion in the number of nodes of the graph with respect to classical approaches.

The next model we consider is Petri nets [22]. A *place-transition net* is a quadruple $N = \langle S, T, F, M_0 \rangle$ consisting of two disjoint sets S and T of places and transitions, of a flow multiset relation $F \subseteq S \times T \cup T \times S$ and of the initial marking $M_0 : S \to \mathbb{N}$ ($\mathbb{N}$ being the set of natural numbers). Note that the flow relation F can be interpreted also as a function weight $W : S \times T \cup T \times S \to \mathbb{N}$.

The dynamic behaviour of nets is defined by the *token game*. A transition $t \in T$ is enabled under a marking M if $\forall s \in S, M(s) \geq W(s,t)$. The occurrence of an enabled transition t produces a new marking M' defined by $\forall s \in S, M'(s) = M(s) - W(s,t) + W(t,s)$. A marking M is called *reachable* if there exists a sequence of transitions that starts from M_0 and leads to M. Also, M is said to be safe if $\forall s \in S, M(s) \leq 1$. A place-transition net is said *safe* if all reachable markings are safe.

In order to characterise runs of nets without abstracting from causal relations between events, *nonsequential processes* were defined [14, 2]. We now introduce the notion of pre- and post-sets. Given an element $x \in S \cup T$, the *pre-set* of x is ${}^\bullet x = \{y \in S \cup T | W(y,x) \neq 0\}$ and the *post-set* of x is $x^\bullet = \{y \in S \cup T | W(x,y) \neq 0\}$. Then, a *process* of a net N is a quadruple $\Pi = \langle B, E, F, p \rangle$ such that $\langle B, E, F \rangle$ is a net with no initial marking, $\langle B \cup E, F \rangle$ is an acyclic graph and $\forall b \in B, |{}^\bullet b| \leq 1$ and $|b^\bullet| \leq 1$. Moreover, $p : B \cup E \to S \cup T$ is a function that maps elements of B to elements of S, elements of E to elements of T and initial places of Π (those with ${}^\bullet b = \emptyset$) to places s such that $M_0(s) \geq 1$. Also, $\forall e \in E, \forall s \in S, W(s, p(e)) = |p^{-1}(s) \cap {}^\bullet e|$ and $W(p(e), s) = |p^{-1}(s) \cap e^\bullet|$. Nielsen, Plotkin and Winskel [21] introduced *occurrence nets* which generalise processes by summarising all the processes of a net as a single net. This is done in by removing the restriction $|b^\bullet| \leq 1$ while preserving reachability by forbidding self-conflict.

We now consider *event structures* [21, 25]. Event structures consist of a set of labelled events together with relations of causality, independence and conflict between them. Indeed it is useful to think of such event structures as having the occurrence net notion of condition, B above, replaced by its abstraction (these relations). Events model the occurrence of actions and are labelled with the action that they represent. Note that an event is quite different from a transition and from a state of the transition system. Indeed, many transitions may represent the same event (e.g. the two α transitions in the event structure originated by the term $\alpha \mid \beta$ represent the same event). More formally, a (prime) event structure is a triple $ES = \langle E, \leq, \# \rangle$ consisting of a set E of events that are partially ordered by $\leq$, the causal dependency relation, and a binary, symmetric, irreflexive relation $\# \subseteq E \times E$, the conflict relation which satisfies $\{e' | e' \leq e\}$ is finite and $e \# e' \leq e'' \Rightarrow e \# e''$ for all $e, e', e'' \in E$. Two events, say e and e', are concurrent or independent (written $e \smile e'$) iff they are neither in conflict nor in the causally related.

A *configuration* of an event structure is a subset $x \subseteq E$ which is *conflict-free* ($\forall e, e' \in x, \neg(e \# e')$) and *downward-closed* ($\forall e \in x, e' \in E, e' \leq e \Rightarrow e' \in x$). The computations of a process are the paths in the partial ordered set of the configurations, ordered by inclusion.

The inheritance property of conflict relation in prime event structures implies that if two events are in conflict, then all causal successors of one event are in conflict with all causal successors of the other event. As a consequence, each event is enabled by a unique set of events and this leads in general to duplication of events (and iteratively to asymptotically greater size) . Consider, e.g., the prime event structure associated to the term $(\alpha + \beta); \gamma$. The event γ is duplicated as it can be enabled by two different events.

In order to avoid duplication of events, *flow event structures* have been introduced [3]. A flow event structure is a triple $FES = \langle E, \prec, \# \rangle$ consisting of a denumerable set E of events, a binary, irreflexive relation $\prec \subseteq E \times E$, the flow relation, and a binary, symmetric relation $\# \subseteq E \times E$, the conflict relation. Configurations are defined through *proving sequences*. A proving sequence is a sequence of distinct events $e_1, \ldots, e_n$ such that $\{e_1, \ldots, e_n\}$ is conflict-free and $\forall e_i, e, e \prec e_i \Rightarrow \exists j < i : e_j \prec e_i$ and ($e = e_j$ or $e \# e_j$). Then, a subset of events $x \subseteq E$ is a configuration if there is a proving sequence $e_1, \ldots, e_n$ such that $x = \{e_1, \ldots, e_n\}$. Note that flow event structures may generate self-conflicting events that will never appear in any configuration.

An alternative to flow event structure for avoiding the unique enabling property of prime event structure are *stable event structures* [25]. In these structures the causality or flow relation is substituted by an explicit enabling relation. Thus, a stable event structure is a triple $SES = \langle E, \vdash, \# \rangle$ where E is a set of events, $\# \subseteq E \times E$ is an irreflexive, symmetric relation, the conflict relation, and $\vdash \subseteq 2^E \times E$ is the enabling relation satisfying consistency ($F \vdash e \Rightarrow F \cup \{e\}$ is conflict free) and stability ($F \vdash e$ and $G \vdash e \Rightarrow F \cup G$ is not conflict-free or $F = G$). Note that the stability condition ensures that the causal relation between events is unambiguous. A proving sequence is a sequence of distinct events $e_1, \ldots, e_n$ such that $\{e_1, \ldots, e_n\}$ is conflict-free and $\forall i, \exists F \subseteq \{e_1, \ldots, e_{i-1}\} : F \vdash e_i$. Finally, a subset of the events $x \subseteq E$ is a configuration if there exists a proving sequence $e_1, \ldots, e_n$ such that $x = \{e_1, \ldots, e_n\}$.

The last class of event structures which we consider is that of *bundle event structures* [18]. Here, the flow, causal or enabling relation is replaced by a bundle set $X \mapsto e$. It represents the set of causal condition for an event e. In order to satisfy stability, all events in X must be pairwise in conflict. Formally, a bundle event structure is a triple $BES = \langle E, \mapsto, \# \rangle$ where E is a set of events, $\# \subseteq E \times E$ is an irreflexive, symmetric relation, the conflict relation, and $\mapsto \subseteq 2^E \times E$ is the bundle set satisfying $X \mapsto e \Rightarrow \forall e, e' \in X : (e \neq e' \Rightarrow e \# e')$. A proving sequence is a sequence of distinct events $e_1, \ldots, e_n$ such that $\{e_1, \ldots, e_n\}$ is conflict-free and $X \mapsto e_i \Rightarrow \{e_1, \ldots, e_{i-1}\} \cap X \neq \emptyset$. Finally, a subset of events $x \subseteq E$ is a configuration if there exists a proving sequence $e_1, \ldots, e_n$ such that $x = \{e_1, \ldots, e_n\}$.

As a last remark, note that all the above classes of event structure may be equipped with a function which labels the events with the actions that they represent.

Note that all the mentioned models may be viewed as algebras (possibly containing infinite terms). Thus, in order to generalise our approach to the complexity of semantic models, we assume that the meta-language into which we map the object-language is an algebra.

Although we rarely spell this out explicitly, it is necessary to regard a meta-language more intensionally than usual and to include its representation as a data-structure. Thus we treat changing the representation of a meta-language as changing the meta-language. Such fine discrimination is necessary to avoid specious claims that *all* semantic functions are linear in size *via* the claim in that they all have a linear size representation of a given object-language term, *viz* itself—see section 2.2.

1.3 Denotational semantics

We wish to cast our net wide in order to include many interpretation[3] mechanisms (including simple translation) as semantic formalisms. We accordingly define a *denotational semantics* to be a 4-tuple $(\mathcal{S}, \mathcal{M}, \mathcal{E}_{i\in I}, \equiv)$ where

- $\mathcal{S}$, the *object-language*, is a free algebra, possibly many-sorted.[4] Sorts of $\mathcal{S}$ are referred to as *syntactic categories*. Identity, $=$, is the usual syntactic identity on free algebras.
- $\mathcal{M}$, the *meta-language*, is an algebra. Typically it is less well-structured that $\mathcal{S}$, for example we might wish to consider $\mathcal{S}$ being Pascal and $\mathcal{M}$ being a set of flowcharts. We allow $\mathcal{M}$ to contain infinite terms in the style common in algebraic semantics [15]. Various CCS programs require infinite event structures or Petri-nets to represent them. It is convenient to require that the operators of $\mathcal{M}$ partition into free constructors which generate the algebra and unfree operators which are required to provide maps from free parts to free parts (compare the idea of constructors and user-defined functions in languages like ML). We see examples of this below (interpretation of *fix* or *par* for synchronisation trees). Identity, $=$, is the usual algebraic equality of terms. This is syntactic identity for the case of a free algebra but also takes into account equations defining any unfree constructors.
- $\mathcal{E}_{i\in I} : \mathcal{S} \to \mathcal{M}$ is an indexed family of *semantic functions*. These are required to be defined by (mutual) primitive recursion. Typically there is one (or more) semantic function for each syntactic category.[5]

[3] In the sense of logic, not an SECD-style interpreter.

[4] It is possible to extend this to allow it (as for the meta-language $\mathcal{M}$) to be a continuous free algebra. The complexity behaviour of limits is defined to be that of their approximants. This allows one to study the inate expressiveness of classes of event structures, Petri-nets and the like by giving mutual translations. This version of the paper does not study such issues.

[5] It is convenient to allow the notion of *environment* to be handled by this scheme too. Often a semantic function (e.g. for a functional language) would be defined with a type such as

$$\mathcal{E}[\![\cdot]\!] : Expr \to (Var \to Val) \to Val$$

where the various $\to$ have differing meanings which require complicated explanation when our meta-language is first order and so lacks a full exponentiation type. This would also make the primitive recursivity and complexity more clumsy to define. However, any possible environment is conveniently handled here by treating it as an

- $\equiv$ is an equivalence relation on $\mathcal{M}$. It is commonly generated by a reduction relation but here we are merely interested in equivalence. We require $m = m' \Rightarrow m \equiv m'$

Although the object- and meta-languages are seen as being (first-order) algebras we can choose to understand various terms as *binding constructs*—this is common for object languages, but the presence of the equivalence relation, $\equiv$, on $\mathcal{M}$ allows natural encoding of the binding rules (e.g. α- and β-equivalence) there too. Such a treatment would normally be considered lacking the necessary abstract elegance. However, we will justify it by our interest in implementation feasibility and complexity.

To clarify the requirements on $\mathcal{M}$ above, suppose it is a meta-language with an operator $\mathit{fix}\, x.e$. It is possible to regard this operator *either* as a free constructor with equivalence caught by $\equiv$, so that $\mathit{fix}\, x.e \equiv e[\mathit{fix}\, x.e/x] \neq \mathit{fix}\, x.e$; *or* as a unfree operator so that $e[\mathit{fix}\, x.e/x] = \mathit{fix}\, x.e$. The former treatment is common in object languages and the λ-calculus as a meta-language whilst the latter is common in event structure or Petri-nets as semantic meta-language for CCS. These treatments, of course, have very different complexity behaviour.

1.4 Correspondence to operational semantics

Often an object language, $\mathcal{S}$ comes complete with a operational semantics and we wish to ensure the denotational semantics is consistent. For the purposes here, we only require the equivalence relation $\approx$ which holds when two programs $s, s' \in \mathcal{S}$ are operationally indistinguishable (actually in all contexts $C[\,]$). For concurrency purposes, $\approx$ is typically one of the forms of bisimulation (strong, weak, interleaving, step, pomset, causality, history-preserving, locality *etc.*).

Denotational semantics $(\mathcal{S}, \mathcal{M}, \mathcal{E}_{i\in I}, \equiv)$ is *consistent* with $\approx$ (or $\approx$-*respecting*) if for all $s, s' \in \mathcal{S}$ we have

$$\mathcal{E}[\![s]\!] \equiv \mathcal{E}[\![s']\!] \Rightarrow (\forall \mathit{contexts}\ C[\,])(C[s] \approx C[s']).$$

It is *computationally adequate* if

$$\mathcal{E}[\![s]\!] \equiv \mathcal{E}[\![s']\!] \Leftarrow (\forall \mathit{contexts}\ C[\,])(C[s] \approx C[s'])$$

and *fully abstract* if both hold.

---

index to $\mathcal{E}$. Hence we can write (continuing the functional programming example)

$$\begin{array}{l} \mathcal{E}_\rho[\![\cdot]\!] : \mathit{Expr} \to \mathit{Val} \\ \mathcal{E}_\rho[\![x]\!] = \rho(x) \\ \mathcal{E}_\rho[\![\lambda x.e]\!] = \mathit{lam}(\lambda v.\mathcal{E}_{\rho[v/x]}[\![e]\!]) \\ \mathcal{E}_\rho[\![e\ e']\!] = \mathit{app}\langle \mathcal{E}_\rho[\![e]\!], \mathcal{E}_\rho[\![e']\!]\rangle \end{array}$$

Primitive recursivity bans the use of such an index as the main argument to $\mathcal{E}$.

1.5 Examples

1. Take $\mathcal{S}$ to be that subset of Pascal generated by the syntactic categories *Cmd* and *Exp* with their usual BNF definition. $\mathcal{M}$ is the usual typed (with recursive types) λ-calculus with $\mathcal{E}_{Cmd}[\![\cdot]\!]$ and $\mathcal{E}_{Exp}[\![\cdot]\!]$ defined in the usual manner. Equivalence is $\alpha\beta\eta\delta$-equivalence (we need δ-rules for the fixpoint-taking constants as an absolute minimum).
2. Take $\mathcal{S}$ to be the language $\mathcal{L}[\alpha,+,|]$ with syntactic category *Beh* given by $Beh \ni b ::= nil \mid \alpha.b \mid b+b' \mid b \mid b'$ with port names α (we also use β in the example below) but no port co-names. $\mathcal{M}$ can be synchronisation trees, ranged over by t, u. The semantic function is interesting—the prefixing and $+$ operators directly encode as tree-forming operators (i.e. the free constructors $\cdot$ and $\uplus$) whereas the $|$ operator is represented by a non-free operator given by the expansion theorem (this language does not require consideration of port and co-port synchronisation):

$$\begin{array}{l}
[\![nil]\!] = \sum_{\emptyset} \alpha \cdot [\![t]\!] = NIL \\
[\![\alpha.b]\!] = \alpha \cdot [\![b]\!] \\
[\![b+b']\!] = \sum_{i\in I} \alpha_i \cdot [\![t_i]\!] \uplus \sum_{i \in J} \alpha_i \cdot [\![t_i]\!] = \sum_{i\in I\cup J} \alpha_i \cdot [\![t_i]\!] \\
\qquad \text{where } [\![b]\!] = \sum_{i\in I} \alpha_i \cdot [\![t_i]\!] \text{ and } [\![b']\!] = \sum_{i\in J} \alpha_i \cdot [\![t_i]\!] \\
[\![b \mid b']\!] = par([\![b]\!], [\![b']\!]) \\
\qquad \text{where } par(t,u) = \sum_{i\in I} \alpha_i \cdot par(t_i, u) \uplus \sum_{i\in J} \beta_j \cdot par(t, u_j) \\
\qquad \text{with } \sum_{i\in I} \alpha_i \cdot t_i = t \text{ and } \sum_{i\in J} \beta_j \cdot u_j = u
\end{array}$$

   Equivalence $\equiv$ is induced by the strong bisimulation on synchronisation trees (i.e., tree isomorphism with additional axioms like absorption $t \uplus t \equiv t$ and $t \uplus NIL \equiv t$).
3. Take $\mathcal{S}$ to be regular expressions. Two alternatives for $\mathcal{M}$ are sets of strings and finite state automata. The corresponding semantic functions are given by the usual constructions in undergraduate texts. However, the focus we wish to make is that the mapping to FSA's has finitary images and is surjective but has a (relatively) complex equivalence which nevertheless is suitable for algorithms; whereas the mapping to sets of strings has infinitary images but with identity as equivalence between semantic images.

To further bring out the discussion of interpreted *versus* free constructors consider the operator *par* above. As defined *par* is an unfree operator which maps the program $\alpha_1.nil|\ldots|\alpha_n.nil$ onto an exponentially bigger synchronisation tree. On the other hand, treating *par* as a free constructor gives a linear-sized semantic function, but with various identities like that of $par(\alpha \cdot NIL, \beta \cdot NIL)$ and $\alpha \cdot \beta \cdot NIL \uplus \beta \cdot \alpha \cdot NIL$ left to be specified in $\equiv$.

1.6 Observation

What we saw above is that there is a spectrum of semantic models and associated semantic functions and equivalences for a given language, $\mathcal{S}$—even when its ostensible semantics is externally prescribed, for example by a relation $\approx$ on object terms. Denotational semantics respecting $\approx$ can range from (essentially) $\mathcal{S}$ with identity as the semantic function and a complicated equivalence $\approx$ to (essentially) the term model given by $\mathcal{S}/\approx$, the induced map to equivalence classes as semantic function and identity as the equivalence relation.

Note that good (in the sense of being useful for proofs) semantic models are typically neither of the extremes discussed above—consider the λ-calculus for imperative languages. When it comes to implementation, further considerations of representation efficiency come to the fore and we consider these in the next section.

The subject of abstract interpretation (see e.g. [8]) typically studies models which are not $\approx$-respecting but which have the property of being finitely computable.

2 Complexity of a semantic model

Given a language $\mathcal{S}$ and a semantics $(\mathcal{S}, \mathcal{M}, \mathcal{E}, \equiv)$ we say the semantics is *finite* if every term of $\mathcal{S}$ has a finite image in $\mathcal{M}$. We say the semantics is *polynomial* (or indeed any other complexity class) if every term of size n in $\mathcal{S}$ has an image in $\mathcal{M}$ which is of at most polynomial size in n. We say it is *infinite* if any (finite) term has an image of infinite size (e.g. due to expansion of $recP.\alpha.P$ as an infinite Petri-net process). Note that these definitions do not include the size of $\equiv$. For example, taking $\mathcal{S} = \mathcal{M}$ and the identity as a semantic function gives a linear semantics.

Given an object language $\mathcal{S}$, we say that a *semantic model* $\mathcal{M}$ is polynomial (or again any other complexity class) if there exists a polynomial semantic function. To avoid the trivial $O(n)$ semantic function which maps every source term onto itself as meta-language term, this definition is only useful when we have prescribed inequalities to preserve (from axioms or from an operational equivalence $\approx$ as discussed above).

When we restrict attention to semantics which respect strong interleaving bisimulation we find that event structures [21] are quadratic for $\mathcal{L}[\alpha, +, |\,]$, but exponential for $\mathcal{L}[\alpha, \overline{\alpha}, +, |]$. However, flow event structures [3] are polynomial for $\mathcal{L}[\alpha, \overline{\alpha}, +, |\,]$ just as stable event structures [25] and bundle event structures [18] are, even if with a lower degree. Synchronisation trees also with richer labelling (e.g., causal trees [9] and the like) are exponential for $\mathcal{L}[\alpha, |\,]$ and infinite for $\mathcal{L}[\alpha, rec]$. Transition systems (even if with independence relations like in [26, 1, 23]) are exponential for $\mathcal{L}[\alpha, |\,]$, but linear for $\mathcal{L}[\alpha, rec]$. They become infinite for $\mathcal{L}[\alpha, |, rec]$. Causal and locational [5] transitions systems are exponential also for $\mathcal{L}[\alpha, rec]$. This happens because the states are incresed at each unfolding with a reference to the activating transitions or to those occurred in the same site, respectively.

If one is interested in truly concurrent equivalences, the above models have the same complexity as for the interleaving semantics, while compact transition systems [11] are linear for $\mathcal{L}[\alpha, +, |\,]$ and polynomial for $\mathcal{L}[\alpha, \overline{\alpha}, +, |\,]$ (without autoconcurrency).

As a final remark, languages allowing value or name passing such as π-calculus [20] or higher order languages like $CHOCS$ [24] have exponential or infinite representations depending on the size of data domains.

These results are summarized in the table reported in the Conclusions.

2.1 Cartesian closedness implies linear complexity

The following result can justify why typed λ-calculus suffices for a great many applications, including classical programming languages. Christiansen and Jones [6] use this technique to simplify the semantic description language in the CERES compiler generator which uses a form of λ-calculus.

Suppose we have a semantic function $\mathcal{E}[\![\,\cdot\,]\!]$ for a syntactic category in a programming language. Suppose that the meta-language is (rather abusing the term) *cartesian-closed*, i.e. there are notions of abstraction, application, pairing and projection. Its homomorphic (primitive recursive) nature means that we can write the semantics of an operator, binary $+$ say, as

$$\mathcal{E}[\![e_1 + e_2]\!] = \ldots \mathcal{E}[\![e_1]\!] \ldots \mathcal{E}[\![e_1]\!] \ldots \mathcal{E}[\![e_2]\!] \ldots$$

which can be transformed into

$$\mathcal{E}[\![e_1 + e_2]\!] = (\lambda m_1.\lambda m_2.\ \ldots m_1 \ldots m_1 \ldots m_2 \ldots)\ (\mathcal{E}[\![e_1]\!])\ (\mathcal{E}[\![e_2]\!]).$$

Note that the former function produces an asymptotically exponentially larger meta-language meaning than the latter which is always linear.

Similarly, suppose we have two or more mutually recursive semantic functions for a given syntactic category. This time pairing comes to our rescue, and functions such as

$$\begin{array}{l} \mathcal{E}[\![e_1 + e_2]\!] = \ldots \mathcal{E}[\![e_1]\!] \ldots \mathcal{F}[\![e_1]\!] \ldots \mathcal{E}[\![e_2]\!] \ldots \\ \mathcal{F}[\![e_1 + e_2]\!] = \ldots \mathcal{F}[\![e_1]\!] \ldots \mathcal{E}[\![e_2]\!] \ldots \mathcal{F}[\![e_2]\!] \ldots \end{array}$$

can be transformed into

$$\mathcal{G}[\![e_1 + e_2]\!] = \begin{pmatrix} \ldots \pi_1\mathcal{G}[\![e_1]\!] \ldots \pi_2\mathcal{G}[\![e_1]\!] \ldots \pi_1\mathcal{G}[\![e_2]\!] \ldots, \\ \ldots \pi_2\mathcal{G}[\![e_1]\!] \ldots \pi_1\mathcal{G}[\![e_2]\!] \ldots \pi_2\mathcal{G}[\![e_2]\!] \ldots \end{pmatrix}$$

where $(\cdot,\cdot)$ represents pairing and π_i the projection functions. The latter function can then be transformed to linear form (each recursive call occurring at most once) by the technique above. Notional references to $\mathcal{E}$ are replaced by $\pi_1\mathcal{G}$ and to $\mathcal{F}$ by $\pi_2\mathcal{G}$.

2.2 Nature of complexity

We argued, by considering (e.g.) the CCS term $\alpha_1.nil| \ldots |\alpha_n.nil$, that, in general, a given CCS term requires an exponentially larger synchronisation tree to express its behaviour.[6] This is sensible from the semantic viewpoint—non-determinism only clumsily represents concurrency. Indeed, if one is interested in "true concurrency", one can build a tree with only a branch representing all the ones differing only in the order in which concurrent transitions are fired (see, e.g., [11, 7, 13]) thus avoiding the exponential size due to state explosion. However, we must be wary of the delicacy of this argument in complexity terms since the resulting tree seen as a data-structure can merely be claimed to be inefficiently represented. (Compare the correct, but unhelpful, argument that binary search is $O(n)$ instead of $O(\log n)$ in total operations if one is required to use linear linked-lists as the data-structure!) In structural complexity terms, all our semantic functions have linear complexity since there is always a linear size representation of the semantics of an object-language term, *viz* the object-language term itself.

We justify our finer distinction by arguing it is reasonable to treat semantics models as specified by their representation as a given data-structure. Thus in general, synchronisation trees are exponentially less powerful than synchronisation DAG's—consider the synchronisation tree and DAG specified by the CCS$[\alpha, +, \mathit{nonrec}]$ program P_n where

$$\begin{aligned} &P_0 \Leftarrow \mathit{nil} \\ &P_1 \Leftarrow \alpha_1.P_0 + \beta_1.P_0 \\ &\cdots \\ &P_n \Leftarrow \alpha_n.P_{n-1} + \beta_n.P_{n-1}. \end{aligned}$$

From a structural complexity viewpoint these are trivially the same but the tree has 2^n arcs and the DAG $2n$ arcs.

Another important aspect we must take into account is the complexity of deducing a computation of a process (simulation) from its semantic representation. Indeed, a transition system may have an exponential size (if interleaving properties are of interest and thus the compact form is not applicable), but the simulation algorithm is linear. Therefore, if we want an estimate of how good a model is for driving the choice of representations for practical purposes, we cannot abstract from dynamic aspects like simulation. A possibility is to associate with each semantic model the higher complexity between the one of the representation and the one of the simulation.

3 Event structures and Petri nets

Interestingly, acyclic Petri-net models fail to be as expressive as event structures for some languages due to possible exponentially bigger representation of conflict (*via* places instead of a relation): consider the language $\mathcal{L}[\alpha, +, |\,]$, and in particular terms of the form $(\alpha_1.\mathit{nil} \mid \beta_1.\mathit{nil}) + \cdots + (\alpha_n.\mathit{nil} \mid \beta_n.\mathit{nil})$. This gives an event structure $(E, \leq, \#)$ with $2n$ events E, equality for the causality

[6] The best has $n!$ branches. Even using an arbitrary labelled transition system requires 2^n arcs.

relation $\leq$ (i.e. size n) and conflict relation of size $n(n-2)$ relating each event α_i (or β_i) to every event in $E \backslash \{\alpha_i, \beta_i\}$. This is of polynomial size, and it turns out that every term in $\mathcal{L}[\alpha, +, |]$ can be so represented. The smallest bisimilar Petri-net for the above term is exponential. It has 2^n places and $2n$ transitions (corresponding to the above events) and 2^n places. Geometrically, visualise an n-dimensional hypercube with a transition at the centre of every hyperface (dimension $n-1$ hyperplane) and place at every vertex (dimension 0-hyperplane) with an arc leading from every place (at vertex v) to every transition whose hyperface has v in its boundary. Thus each transition has 2^{n-1} pre-places (i.e. half of all places).

Of course, simple event structures are far from perfect in a complexity sense; for example the language $\mathcal{L}[\alpha, +, \mathit{nonrec}]$ has linear complexity with acyclic Petri-nets (or synchronisation DAG's) but exponential complexity for event structures (or synchronisation trees). This time the program P_n where

$$
\begin{aligned}
&P_0 \Leftarrow \mathit{nil} \\
&P_1 \Leftarrow \alpha_1.P_0 + \beta_1.P_0 \\
&\ldots \\
&P_n \Leftarrow \alpha_n.P_{n-1} + \beta_n.P_{n-1}
\end{aligned}
$$

provides a critique of event structures which was resolved by Boudol's flow event structures.

Again, although the net and event structure constructions corresponding to CCS operators are well-known, looking at this issue from a complexity viewpoint forces one to contemplate the above consequences which appear not to be well-known.

These observations can be retrospectively seen as providing the motivation for the search for more implementation-apt models for CCS such as extensions to elementary event structures.

4 No free lunches

It should be clear from the earlier discussions that low complexity and full abstractness are mutually antagonistic. Indeed, if a language has Turing power (for example full CCS) then its operational equivalence $\approx$ is undecidable. Now, this means that any fully abstract semantics (with equality as meta-language equivalence) must be infinite in the sense of translating some finite term to an infinite one.

On the other hand, it *is* worth studying the complexity results for sub-languages because of the light such results cast on expressiveness of object- and meta-language constructs.

The key to effective implementation of proof systems appears to be to choose a semantic model which answers quickly many of the questions which are likely to be posed, but is willing to admit defeat.

5 Conclusions

We have presented the notion of complexity of a semantic function or semantic model which seems important for helping give quantitative classifications of semantic models. We have shown how these can be written in a form which always yields linear complexity for cartesian-closed semantic models.

To conclude, here is a table showing the complexity of several natural (in the sense of preserving as much structure as possible) semantic functions between the various semantic models discussed above:

Object-	meta-language:							
Language	ON	SNP	ES	xES	TS	xTS	CTS	T
ON	$O(n)$	$O(n^2)$						
SNP	$O(n)$	$O(n)$						
$\mathcal{L}[\alpha, +]$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	O(n)	$O(n)$
$\mathcal{L}[\alpha, +, \mid]$	$O(2^n)$	$O(2^n)$	$O(n^2)$	$O(n^2)$	$O(2^n)$	$O(2^n)$	O(n)	$O(2^n)$
$\mathcal{L}[\alpha, +, ;]$	$O(2^n)$	$O(2^n)$	$O(2^n)$	$O(n^2)$	P	$O(2^n)$	P	$O(2^n)$
$\mathcal{L}[\alpha, +, nonrec]$	$O(2^n)$	$O(2^n)$	$O(2^n)$	P	P	P	P	$O(2^n)$
$\mathcal{L}[\alpha, \overline{\alpha}, +, \mid]$	$O(2^n)$	$O(2^n)$	$O(2^n)$	P	$O(2^n)$	$O(2^n)$	P†	$O(2^n)$
$\mathcal{L}[\alpha, \overline{\alpha}, +, \mid, \backslash]$	$O(2^n)$	$O(2^n)$	$O(2^n)$	P	$O(2^n)$	$O(2^n)$	$O(2^n)$	$O(2^n)$
$\mathcal{L}[\alpha, rec]$	∞	∞	∞	∞	$O(n)$	∞	O(n)	∞
$\mathcal{L}[\alpha, \mid, rec]$	∞	∞	∞	∞	∞	∞	∞	∞

† = only in the absence of autoconcurrency.
Complexity classes: P (polynomial), ∞ (infinite).
Abbreviations: ON (occurrence net), SNP (set of net processes); ES (event structures as above), xES (flow, stable or bundle event structures); TS (transition systems with or without independence); xTS (causal or locational transition systems); CTS (compact transition systems); T (various kind of trees).

Note that an embedding (e.g. of synchronisation trees into event structures) leads to $O(n)$ complexity.

Acknowledgments

We are grateful to the EU ESPRIT BRA grant 8130 "LOMAPS" for financial support. The second author benefitted from support from LIX (Laboratoire d'Informatique, Ecole Polytechnique) whilst on sabbatical leave there.

References

1. M.A. Bednarczyk. *Categories of Asynchronous Transition Systems.* PhD thesis, University of Sussex, 1988.
2. E. Best and R. Devillers. Sequential and concurrent behaviour in Petri net theory. *Theoretical Computer Science*, 55(1):87–136, 1987.

3. G. Boudol. Flow event structures and flow nets. sémantique du parallelisme, i. guessarian (ed.), lncs 469, 1990. In I. Guessarian, editor, *Semantics of systems of concurrent processes, Proc. 18ème école de printemps d' informatique théorique, LNCS 469*. Springer Verlag, 1990.
4. G. Boudol and I. Castellani. A non-interleaving semantics for CCS based on proved transitions. *Foundamenta Informaticae*, XI(4):433–452, 1988.
5. G. Boudol, I. Castellani, M. Hennessy, and A. Kiehn. A theory of processes with localities. *Theoretical Computer Science*, 114, 1993.
6. H. Christiansen and N.D. Jones. Control-flow treatment in a simple semantics-directed compiler generator. In D. Björner, editor, *IFIP WG 2.2:Formal description of programming concepts II*. North-Holland, 1983.
7. M. Clegg and A. Valmari. Reduced labelled transition systems save verification effort. In *Proceedings of CONCUR'91, LNCS 527*. Springer-Verlag, 1991.
8. P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, pages 511 – 547, 1992.
9. Ph. Darondeau and P. Degano. Causal trees. In *Proceedings of ICALP'89, LNCS 372*, pages 234–248. Springer-Verlag, 1989.
10. P. Degano and C. Priami. Proved trees. In *Proceedings of ICALP'92, LNCS 623*, pages 629–640. Springer-Verlag, 1992.
11. P. Degano and C. Priami. A compact representation of finite-state processes. Technical Report LOMAPSDIPISA2, LOMAPS Project, 1993. Submitted for publication.
12. G. Ferrari, R. Gorrireri, and U. Montanari. An extended expansion theorem. In *Proceedings of TAPSOFT'91, LNCS 431*, pages 162–176. Springer-Verlag, 1991.
13. P. Godefroid and P. Wolper. Using partial orders for the efficient verification of deadlock freedom and safety properties. In *Proceedings of CAV'91, LNCS 575*, pages 332–342. Springer-Verlag, 1991.
14. U. Goltz and W. Reisig. The non-sequential behaviour of petri nets. *Information and Computation*, 57:125–147, 1983.
15. I. Guessarian. Algebraic semantics. In *LNCS 99*. Springer Verlag, 1981.
16. M. Hennessy. *Algebraic Theory of Processes*. MIT Press, 1988.
17. R.M. Keller. Formal verification of parallel programs. *Communications of the ACM*, 19:371–384, 1976.
18. R. Langerak. *Transformations and semantics for LOTOS*. PhD thesis, Twente University, 1992.
19. R. Milner. *Communication and Concurrency*. Prentice-Hall International, London, 1989.
20. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes (I and II). *Information and Computation*, 100(1):1–77, 1992.
21. M. Nielsen, G. Plotkin, and G. Winskel. Petri nets, event structures and domains. *Theoretical Computer Science*, pages 85–108, 1981.
22. W. Reisig. Petri nets: An introduction. In *EATCS Monographs*. Springer-Verlag, 1985.
23. A. Stark. Concurrent transition systems. *Theoretical Computer Science*, pages 221–269, 1989.
24. B. Thomsen. *Calculi for Higher Order Communicating Systems*. PhD thesis, Imperial College - University of London, 1990.
25. G. Winskel. Event structures for CCS and related languages. In *Proceedings of ICALP'82, LNCS 140*, pages 561–576. Springer-Verlag, 1982.

26. G. Winskel and M. Nielsen. Models for concurrency. Technical Report DAIMI-PB-429, Computer Science Dept., Aarhus University, 1992. To appear in Handbook of Logic in Computer Science.