STATIC ANALYSIS OF VALUE-PASSING PROCESS CALCULI

(Extended Abstract)

DAVID CLARK, LINDSAY ERRINGTON and CHRIS HANKIN
Department of Computing, Imperial College
180 Queen's Gate, London SW7 2BZ, United Kingdom
E-mail: {djc2, le, clh}@doc.ic.ac.uk

ABSTRACT

Standard approaches to providing a semantics for value passing process calculi involve an expansion into a basic calculus which only involves pure synchronisation. When the semantics is intended to provide a basis for static analysis of properties of the values, this expansion leads to an unacceptable loss of information. In this paper we present a model of value passing process calculi based on synchronisation trees which gives a direct treatment of value passing. We present an abstract interpretation which discovers the same information about programs as Schreiber's flow analysis.

1 Introduction

Schreiber has examined the problems of specifying and implementing communications protocols, using the value passing version of Milner's Calculus of Communicating Systems (CCS) to do both [7]. The syntactic form of CCS is not a convenient setting for the algorithms he wishes to develop so he abstracts away from it slightly and instead uses a sequential semantic representation of a useful class of CCS expressions. The class can loosely be described as finite state and finite branching. Schreiber calls the representation that he uses parametrised graphs - graphs in which each node represents a value-passing agent and is associated with the details of the agent's parameters while each edge is labelled with the transition that it represents. The advantage is that parallel operators in the original syntax "disappear" into the various states and so the representation admits a solely sequential treatment just as in a labelled transition system. Since the original CCS expression is finite state and finite branching, the graph obtained for a process expression has a finite number of nodes and edges and thus a finite representation.

A particular implementation of a given specification is verified by finding a bisimulation between the parametrised graphs representing the two. Since the equivalence may, in general, depend on the equality of functions or on conditions on the parameters for the expressions represented by the graphs, an algorithm attempting to define the bisimulation between the specification and the implementation will generate a set of proof obligations expressed in terms of the underlying value language. In order to minimise the amount of effort needed to meet these proof obligations, it may be desirable to eliminate certain variables from the value passing calculus. For example, an output parameter on a given channel may be involved in a multiplication:

$$\text{agent} \quad B(y : \{1, ..., 10\}) = a!(y \times 2).\text{nil}$$

Milner [6], demonstrates that the relationship between value-passing CCS and (so-called) basic CCS is that the former is a contracted version of the latter. He shows that processes in value-passing CCS may be translated to basic CCS through expanding each channel of a given sort to a number of channels, one for each element in the sort. Thus the process in the example above would become a family of processes:

$$\text{agent} \quad B_i = a_i!(i \times 2).\text{nil} \quad \text{where} \quad i \in \{1, ..., 10\}$$

Schreiber can eliminate undesirable variables in a process for parametrised graphs which is analogous to Milner's expansion for value passing CCS. However, having chosen to eliminate a particular variable by expansion on a channel associated with it there may be other variables dependent on this one which will also need to be expanded. As a result it is necessary to do a kind of dependency analysis on a graph in order to expand parameters (and channels) in a consistent way. The analysis is developed as a static analysis which finds a consistent parameter marking containing a given initial marking (which marks the parameters and variables that we wish to avoid in the proof obligations) .

One of our objectives is to define an abstract interpretation equivalent to Schreiber's analysis. A prerequisite for this work is to have an adequate semantics for value passing CCS. Since Schreiber's analysis detects dependencies between variables, it seems to be important that the semantics should preserve variables. This paper develops a denotational model for value passing process calculi in which variables are not eliminated. We define a generalisation of synchronisation trees where constraints on variables are associated with transitions.

In Section 2 we introduce our notation. Synchronisation trees are introduced in Section 3; in this section we provide a model for value passing process calculi which is based on an extended definition of synchronisation trees in which the values are explicitly represented. An abstract interpretation is presented in Section 4. We conclude in Section 5.

2 A value passing process calculus

We follow Winskel [8, 11] and define a generic process calculus and later show CCS to be an instance of it. It is a value passing language and we assume the value language is defined by means of a first-order term algebra on top of which we build a process language with the usual operators for prefixing, sum and parallel composition.

More formally, the syntax and semantics of the value language are characterised by a triple $\mathcal{L} = (\Sigma, A, X)$ where Σ is a signature, A is a Σ-algebra and X is a set of variables sorted according to the signature. As is standard, we write $T(\Sigma, X)$ for the corresponding term algebra.

We will need a few more definitions from universal algebra which we include here in order to fix our notation.

First, term equations have the form $t_1 = t_2$ for terms $t_1, t_2 \in T(\Sigma, X)$. The well-formed formulae, $\mathit{WFF}(\Sigma)$, are the set of all finite conjunctions of term equations.

A valuation with respect to a Σ-algebra A is a function $\nu : X \to A$ which assigns a value to each variable. A valuation ν can be readily extended to apply to terms and programs and such functions will be denoted by ν as well.

For any Σ-algebra A, valuation $\nu : X \to A$, and formula $\phi \in \mathit{WFF}(\Sigma)$, the *satisfaction relation* written $A,\nu \models \phi$ is defined inductively as follows:

- $A,\nu \models t = t' \iff \nu(t) = \nu(t')$
- $A,\nu \models (\phi \wedge \psi) \iff A,\nu \models \phi$ and $A,\nu \models \psi$

Now we define the process calculus. A program is a finite set of mutually recursive definitions of the form

$$P_i(x_1, \ldots, x_{n_i}) \stackrel{\text{def}}{=} e_i$$

together with an initial call. The syntax conforms to the grammar shown in Fig. 1 where $b \in \mathit{WFF}(\Sigma)$, $\vec{t} \in T(\Sigma, X)^*$, $1 \leq k \leq n$, Λ is a set of labels and $\delta \in \mathit{Act}$ is an action.

The choice of the set Act partly determines the particular process calculus one wishes to model. We list below the choices for basic CCS and value passing CCS.

$$\begin{array}{lcl} \mathit{Act}_{bCCS} & = & L \cup \overline{L} \cup \{\tau\} \\ \mathit{Act}_{CCS} & = & (L \times X) \cup (\overline{L} \times T(\Sigma, X)) \cup \{\tau\} \end{array}$$

In both cases L is a set of labels or port names and $\overline{L}$ is the set of complementary (output) ports. Thus an input action in value passing CCS, for example, consists of a label and variable while an output action consists of a co-label and a term. The symbol τ represents the silent action.

Omitting the conditional from the syntax and requiring that the vector of terms in a call, $\vec{t}$, and the vector of variables in each process definition, $\vec{x}$,

$$
\begin{array}{lcll}
\mathit{prog} & ::= & P_k(\vec{t}) \ \mathsf{where}\ \{P_i(\vec{x}) \stackrel{\mathrm{def}}{=} e_i\}_{i=1}^{n} & \\
e & ::= & \mathsf{nil} & \\
 & | & \delta \rhd e & \text{prefixing} \\
 & | & P_j(\vec{t}) & \text{call} \\
 & | & e_1 + e_2 & \text{sum} \\
 & | & e_1 \parallel e_2 & \text{parallel composition} \\
 & | & e \setminus \Lambda & \text{restriction} \\
 & | & \mathsf{if}\ b\ \mathsf{then}\ e & \text{conditional}
\end{array}
$$

Figure 1: Syntax of the value passing calculus

be empty yields a non-value passing or basic process calculus. We assume the existence of a function $Tr(\cdot)$ which maps a program in the value passing calculus into an equivalent program in the basic calculus using the translation given by Milner [6].

3 Synchronisation trees

Synchronisation trees were introduced by Milner for defining the semantics of basic CCS [5]. For Milner a synchronisation tree is a rooted, possibly infinite tree where each edge is labelled by an element from a set of actions *Act*. The nodes of a tree represent states and the edges are the observable actions a process can perform from a given state.

We adopt the definitions of Nielsen and Winskel [9] in which they define synchronisation trees as particular instances of transition systems.

Definition 1 *A* labelled transition system *is a structure* $(S, i, Act, Tran)$ *where*

- S *is a set of states with initial state* i.
- Act *is a set of labels or actions,*
- $Tran \subseteq S \times Act \times S$ *is the transition relation*

Definition 2 *A* synchronisation tree *is a transition system* $(S, i, Act, Tran)$ *where*

- *every state is reachable,*
- *if* $s \xrightarrow{a_1} \cdots \xrightarrow{a_n} s$ *for a string of labels* $a_1, \ldots, a_n$*, then the string is empty (i.e. the transition system is acyclic),*
- $s' \xrightarrow{a} s \ \wedge\ s'' \xrightarrow{b} s \implies a = b \ \wedge\ s' = s''$

Before discussing synchronisation tree models of value-passing calculi, we review their application to basic calculi.

3.1 Synchronisation trees for basic process calculi

Let $ST(Act)$ be the class of synchronisation trees with transitions labelled from the set Act and let ϵ denote the empty tree. The operations prefixing, sum and restriction for synchronisation trees over a label set Act are shown in Fig. 2. The inference rules are essentially the same as those defining the operational semantics of CCS, only here they are operators on synchronisation trees.

Prefixing $$\frac{}{\delta \rhd p \xrightarrow{\delta} p} \quad \delta \in Act,\ p \in ST(Act)$$

Sum $$\frac{p \xrightarrow{\delta} p'}{p + q \xrightarrow{\delta} p'} \qquad \frac{q \xrightarrow{\delta} q'}{p + q \xrightarrow{\delta} q'}$$

Restriction $$\frac{p \xrightarrow{\delta} p'}{p \setminus R \xrightarrow{\delta} p' \setminus R} \quad \delta \notin R$$

Figure 2: Operators on synchronisation trees

Next we give a schematic semantics for the basic calculus. In the definition in Fig. 3 the domain Act and the parallel composition operator $\|$ are left unspecified. Instantiating the equations with different interpretations $(Act, \|$ $)$ will yield semantic definitions for different languages. Parameterising the semantics in this way has been adapted from Winskel [11, 8, 10].

The semantics of basic CCS are given by the equations in Fig. 3 with the interpretation $(Act_{bCCS}, \|_{bCCS})$ where:

$$\frac{p \xrightarrow{\delta} p'}{p \|_{bCCS} q \xrightarrow{\delta} p' \|_{bCCS} q} \qquad \frac{q \xrightarrow{\delta} q'}{p \|_{bCCS} q \xrightarrow{\delta} p \|_{bCCS} q'}$$

$$\frac{p \xrightarrow{l} p' \qquad q \xrightarrow{\bar{l}} q'}{p \|_{bCCS} q \xrightarrow{\tau} p' \|_{bCCS} q'}$$

3.2 Synchronisation trees for value passing calculi

Although the emphasis of his paper is non-value passing calculi, Winskel also includes a synchronisation algebra for value passing CCS. He does this in much the same way that Milner translates the value passing CCS into basic CCS [6] - by expanding the tree and eliminating variables.

As discussed earlier, we seek a model which preserves variables and this requires a generalisation of synchronisation trees. We replace the alphabet used

$$
\begin{array}{l}
Env = PName \to ST(Act) \\
\\
\mathbf{P}_b : Prog \to ST(Act) \\
\mathbf{D}_b : Def \to Env \to Env \\
\mathbf{E}_b : C \to Env \to ST(Act)
\end{array}
$$

$$
\begin{array}{lcl}
\mathbf{P}_b \; [\![P_k \; \mathsf{where} \; \{d_i\}_{i=1}^n]\!] & = & (\mathbf{fix} \bigsqcup_{i=1}^n (\mathbf{D}_b \; [\![d_i]\!])) \; P_k \\
\mathbf{D}_b \; [\![P_i \stackrel{\mathrm{def}}{=} e_i]\!] \; \rho \; P & = & \begin{cases} \mathbf{E}_b \; [\![e_i]\!] \; \rho & \text{if } P = P_i \\ \epsilon & \text{otherwise} \end{cases} \\
\mathbf{E}_b \; [\![\mathsf{nil}]\!] \; \rho & = & \epsilon \\
\mathbf{E}_b \; [\![\delta \triangleright e]\!] \; \rho & = & \delta \triangleright (\mathbf{E}_b \; [\![e]\!] \; \rho) \\
\mathbf{E}_b \; [\![P_j]\!] \; \rho & = & \rho \; P_j \\
\mathbf{E}_b \; [\![e_1 + e_2]\!] \; \rho & = & (\mathbf{E}_b \; [\![e_1]\!] \; \rho) + (\mathbf{E}_b \; [\![e_2]\!] \; \rho) \\
\mathbf{E}_b \; [\![e_1 \parallel e_2]\!] \; \rho & = & (\mathbf{E}_b \; [\![e_1]\!] \; \rho) \parallel (\mathbf{E}_b \; [\![e_2]\!] \; \rho) \\
\mathbf{E}_b \; [\![e \setminus \Lambda]\!] \; \rho & = & (\mathbf{E}_b \; [\![e]\!] \; \rho) \setminus \Lambda
\end{array}
$$

Figure 3: Synchronisation tree semantics for a basic process calculus.

for labelling by a richer structure. As before, edges of a synchronisation tree are labelled with actions, only now, actions may include variables and terms. Moreover, we must also record information about variable bindings. For example, in CCS if one process performs an action $a(x)$ while another performs the action $\overline{a}(h(y, z))$, their parallel composition should have a τ transition. However, we must record that from that point on, the variable x is assigned the value $h(y, z)$. We should also keep in mind that y and z may be free variables.

One way to handle such a binding is to form the substitution $x \mapsto h(y, z)$ and apply it to all occurrences of x in the tree representing the parallel composition. This mimics the rule for synchronisation in the operational semantics of CCS. Alternatively, rather than apply the substitution, we could record it in the tree representing the parallel composition by pairing it with the τ in the transition. This way, bindings become explicit in the model and potential subjects of an abstract interpretation.

However, rather than substitutions, it is convenient to record variable binding information with term equations. Thus, in addition to labelling each transition with an action, we attach a conjunction of term equations such that the labelling set becomes $Act \times WFF(\Sigma)$. Formulae attached to transitions are interpreted as constraints on variables or, more precisely, they impose constraints on the set of valuations in which the transition can take place. For the example above, the transition in the parallel composition would be labelled

$(\tau, x = h(y, z))$.

We use formulae rather than substitutions for two reasons. First it is applicable to more process calculi. In CSP, for example, there are no τ transitions so it is not clear to which transition one should attach a substitution. The second reason for using such formulae is that they provide a simple way of handling conditionals. To construct a tree for if b then e, we conjoin the condition b to all the initial transitions of the tree modelling e. As expected, a branch can be taken only when the condition is satisfied.

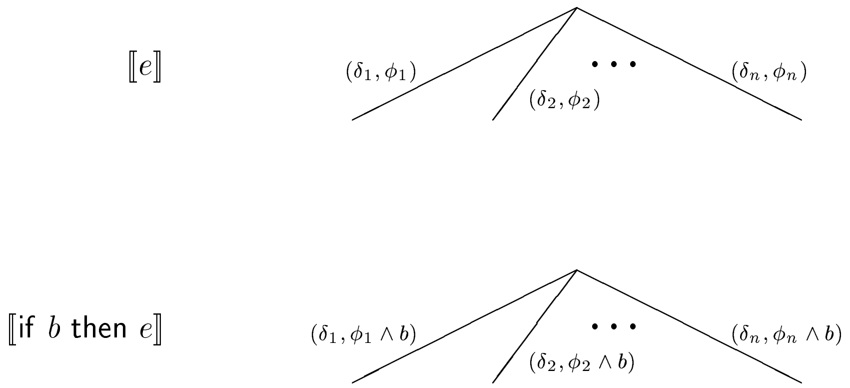

Figure 4: Synchronisation trees for conditionals

Note that variable bindings come about not only by synchronisations but also more mundanely when one process makes a call to another. Again, one option is to substitute the call arguments for the variables in the declaration. However, following our policy of making variable bindings explicit in the model, we again use formulae. Thus, we have a uniform way of recording within the model all constraints on variables.

Fig. 5 shows a schematic semantics. The underlined names are parameters to the specification. The parameters $\underline{CT}$ and $\underline{ST}$ represents the "constraints" and "synchronisation trees" respectively; their use will become clear below.

The semantics of value passing CCS are given by the interpretation shown in Fig. 6.

The intuition for the parallel composition operator is that a synchronisation (transition) can take place only when the labels are complementary pairs and when the individual transitions can take place (i.e. when both ϕ_0 and ϕ_1 are satisfied). Further the valuations are constrained then and thereafter to those which satisfy the equations in which the variables on the input channel are equated with the corresponding terms on the output channel.

For synchronisation trees labelled with actions and formulae, we can define a bisimulation relation. Let Ω denote the set of all valuations on the Σ-algebra

$Env = PName \to T(\Sigma, X)^* \to \underline{CT} \to \underline{ST}$

$\mathbf{P} : Prog \to \underline{ST}$
$\mathbf{D} : Def \to Env \to Env$
$\mathbf{E} : C \to Env \to \underline{CT} \to \underline{ST}$

$$
\begin{array}{lcl}
\mathbf{P}\ [\![P_k(\vec{t})\ \textsf{where}\ \{d_i\}_{i=1}^n]\!] & = & (\mathbf{fix} \bigsqcup_{i=1}^n (\mathbf{D}\ [\![d_i]\!]))\ P_k\ \vec{t}\ \bot_{\underline{CT}} \\
\mathbf{D}\ [\![P_i(\vec{x}) \stackrel{\text{def}}{=} e_i]\!]\ \rho\ P\ \vec{t}\ \phi & = & \begin{cases} \underline{restrict}(\mathbf{E}\ [\![e_i']\!]\ \rho\ \underline{comb}(\underline{con}(\vec{x}', \vec{t}), \phi)) & \text{if } P = P_i \\ \bot_{ST} & \text{otherwise} \end{cases} \\
\mathbf{E}\ [\![\textsf{nil}]\!]\ \rho\ \phi & = & \bot_{ST} \\
\mathbf{E}\ [\![\delta \triangleright e]\!]\ \rho\ \phi & = & (\delta, \phi) \underline{\triangleright} (\mathbf{E}\ [\![e]\!]\ \rho\ \bot_{\underline{CT}}) \\
\mathbf{E}\ [\![P_j(\vec{t})]\!]\ \rho\ \phi & = & \rho\ P_j\ \vec{t}\ \phi \\
\mathbf{E}\ [\![e_1 + e_2]\!]\ \rho\ \phi & = & (\mathbf{E}\ [\![e_1]\!]\ \rho\ \phi) \underline{+} (\mathbf{E}\ [\![e_2]\!]\ \rho\ \phi) \\
\mathbf{E}\ [\![e_1 \parallel e_2]\!]\ \rho\ \phi & = & (\mathbf{E}\ [\![e_1]\!]\ \rho\ \phi) \underline{\parallel} (\mathbf{E}\ [\![e_2]\!]\ \rho\ \phi) \\
\mathbf{E}\ [\![e \setminus \Lambda]\!]\ \rho\ \phi & = & (\mathbf{E}\ [\![e]\!]\ \rho\ \phi) \underline{\setminus} \Lambda \\
\mathbf{E}\ [\![\textsf{if}\ b\ \textsf{then}\ e]\!]\ \rho\ \phi & = & \mathbf{E}\ [\![e]\!]\ \rho\ \underline{combif}(\phi, b)
\end{array}
$$

Where:

- $(e_i', \vec{x}') = \underline{new}(e_i, \vec{x})$

Figure 5: A Schematic Semantics

A and variable set X. Given a set of valuations V and a formula $\phi \in WFF(\Sigma)$, define $V \lceil \phi = \{\nu \in V \mid A, \nu \models \phi\}$.

Definition 3 *A* value passing bisimulation *is a binary relation* $\sim \subset ST(Act \times WFF(\Sigma))^2$ *such that* $p \sim q \iff (p, \Omega) \sim' (q, \Omega)$ *where* $(p, V_p) \sim' (q, V_q)$ *implies*

- $p \xrightarrow{(\delta,\phi)} p' \implies \forall \nu_p \in V_p \lceil \phi,\ \exists q', \beta, \psi, \nu_q \in V_q \lceil \psi\ .\ q \xrightarrow{(\beta,\psi)} q' \wedge \nu_p(\delta) = \nu_q(\beta) \wedge (p', V_p \lceil \phi) \sim' (q', V_q \lceil \psi)$
- $q \xrightarrow{(\beta,\psi)} q' \implies \forall \nu_q \in V_q \lceil \psi,\ \exists p', \delta, \phi, \nu_p \in V_p \lceil \phi\ .\ p \xrightarrow{(\delta,\phi)} p' \wedge \nu_p(\delta) = \nu_q(\beta) \wedge (p', V_p \lceil \phi) \sim' (q', V_q \lceil \psi)$

3.3 Relating the semantic definitions

We show the correspondence between the synchronisation tree constructed for a value passing CCS program and the tree constructed after that program is first translated into basic CCS. The correspondence takes the form of a bisimulation between the respective trees.

Let $Em(\cdot) : ST(Act_{bCCS}) \to ST(Act_{CCS} \times WFF(\Sigma))$ be the function which embeds a synchronisation tree without formulae to a synchronisation tree la-

$$\begin{array}{rcl}
CT & = & \mathit{WFF}(\Sigma) \\
ST & = & ST(Act \times \mathit{WFF}(\Sigma)) \\
\\
restrict & = & id \\
comb & = & \wedge \\
\\
con\ (x_1, \ldots, x_n)\ (t_1, \ldots, t_n) & = & (x_1 = t_1) \wedge \ldots \wedge (x_n = t_n) \\
\\
new & & \text{generates new variable names} \\
\\
\triangleright, +, \backslash & & \text{are as defined earlier for synchro-} \\
& & \text{nisation trees} \\
\\
combif & = & \wedge
\end{array}$$

$$\frac{p \xrightarrow{(\delta,\phi)} p'}{p \| q \xrightarrow{(\delta,\phi)} p' \| q} \qquad \frac{q \xrightarrow{(\delta,\phi)} q'}{p \| q \xrightarrow{(\delta,\phi)} p \| q'}$$

$$\frac{p \xrightarrow{((a,\vec{x}_0),\phi_0)} p' \quad q \xrightarrow{((\overline{a},\vec{t}_1),\phi_1)} q'}{p \| q \xrightarrow{(\tau,\phi_0 \wedge \phi_1 \wedge con(\vec{x}_0,\vec{t}_1))} p' \| q'}$$

Figure 6: Semantics for value-passing CCS

belled with formulae by mapping each transition $p \xrightarrow{\delta} p'$ to $Em(p) \xrightarrow{(\delta,\mathsf{true})} Em(p')$.

Theorem 1 *For all programs P,* $\mathbf{P}\ [\![P]\!] \sim Em(\mathbf{P}_b\ [\![Tr(P)]\!])$

4 The Abstract Interpretation

4.1 The analysis

For our abstract interpretation we intend to follow the approach taken by Marriott *et.al.* [4] for the abstract interpretation of logic programs. There, as here, constraints on variables are recorded in the concrete semantics using term equations. Using an idea developed by Dart [2, 3], they define an abstract interpretation for groundness analysis of logic variables in which term equations are abstracted by propositional formulae. For example, the equation $x = h(y, z)$ is abstracted by $x \leftrightarrow (y \wedge z)$. The intuition is that if the variable x is ground then so too are the variables y and z and vice versa. We intend to apply the same abstract domain to perform dependency analysis on program variables and use the dependency information to determine minimal consistent parameter expansions.

Both well-formed formulae and synchronisation trees are abstracted by pairs; the first component is a proposition, the second a set of live variables. Propositions are conjunctions of dependencies of the form $var_1 \leftarrow var_2$ or $var_1 \leftrightarrow var_2$; we find it convenient to represent such propositions by sets of dependencies. The ordering on propositions is then just the standard subset ordering. The ordering on the second component is also subset and the ordering on pairs, $\leq$, is pointwise.

The abstract interpretation is presented in Fig. 7.

$$
\begin{array}{rcl}
CT^\sharp & = & Prop \times \mathcal{P}(X) \\
ST^\sharp & = & Prop \times \mathcal{P}(X) \\
\\
restrict^\sharp(P,V) & = & (\{x \leftarrow y \mid (x \leftarrow y) \sqsubseteq P, x, y \in V \cup L \cup \overline{L}\}, V) \\
comb^\sharp((\phi_1,\sigma_1),(\phi_2,\sigma_2)) & = & (\phi_1 \cup \phi_2, \sigma_1 \cup \sigma_2) \\
\\
con^\sharp(\vec{x},\vec{t}) & = & (\bigcup_i(\{x_i \leftarrow y \mid y \in Vars(t_i)\}), \bigcup_i Vars(t_i)) \\
 & & \text{where } Vars \text{ returns the set of live variables} \\
 & & \text{occuring in its argument} \\
\\
new^\sharp & = & id \\
\\
(a(x),(\phi_1,\sigma_1)) \rhd^\sharp (\phi_2,\sigma_2) & = & (\phi_1 \cup \phi_2 \cup \{x \leftarrow a\}, \sigma_1 \cup \sigma_2) \\
(\overline{a}(t),(\phi_1,\sigma_1)) \rhd^\sharp (\phi_2,\sigma_2) & = & (\phi_1 \cup \phi_2 \cup \{a \leftarrow x \mid x \in Vars(t)\}, \sigma_1 \cup \sigma_2) \\
(\tau,(\phi_1,\sigma_1)) \rhd^\sharp (\phi_2,\sigma_2) & = & (\phi_1 \cup \phi_2, \sigma_1 \cup \sigma_2) \\
\\
(\phi_1,\sigma_1) +^\sharp (\phi_2,\sigma_2) & = & (\phi_1 \cup \phi_2, \sigma_1 \cup \sigma_2) \\
\\
(\phi_1,\sigma_1) ||^\sharp (\phi_2,\sigma_2) & = & (\phi_1 \cup \phi_2 \cup \bigcup_{a \in L \cup \overline{L}}\{con^\sharp(\vec{x},\vec{y})\}, \sigma_1 \cup \sigma_2) \\
 & & \text{where } \vec{x} \text{ is a sequence of variables such that} \\
 & & x \leftarrow a \text{ is in } \phi_i \text{ and } \vec{y} \text{ is a sequence of variables} \\
 & & \text{such that } a \leftarrow t \text{ is in } \phi_i \\
\\
(\phi,\sigma) \backslash^\sharp \Lambda & = & (\phi \setminus \Lambda, \sigma) \\
\\
combif^\sharp((\phi,\sigma),b) & = & (\phi, \sigma \cup Vars(b))
\end{array}
$$

Figure 7: The Abstract Interpretation

4.2 Correctness

We start by defining an abstraction map, α_1, between sets of well-formed formulae and the abstract domain; this map is not of much interest in itself but it is useful in defining the abstraction map on synchronisation trees.

Definition 4 (Abstraction of $WFF(\Sigma)$)

$$\alpha_1 : \mathcal{P}(WFF(\Sigma)) \rightarrow Prop \times \mathcal{P}(X)$$

$$\begin{array}{rcl} \alpha_1(\phi) & = & \{(\bigcup_i(\{x \leftarrow y \mid x \in Vars(unnew(t_{i_1})), y \in Vars(unnew(t_{i_2}))\}), \\ & & \bigcup_i(Vars(unnew(t_{i_1})) \cup Vars(unnew(t_{i_2})))) \mid (\bigwedge_i t_{i_1} = t_{i_2}) \in \phi\} \end{array}$$

Notice that the abstract interpretation does not generate new copies of variables. The function *unnew* returns a term where instances of variables are replaced by the original variable. This process is reminiscent of the abstraction of locations found in some compile-time garbage collection analyses.

Proposition 1 *α_1 is monotonic and surjective.*

Hence we can induce a concretisation function in the standard way:

$$\gamma_1(P) = \bigcup\{\phi \mid \alpha_1(\phi) \leq P\}$$

Definition 5 (Abstraction of $ST(Act \times WFF(\Sigma))$)

$$\alpha_2 : ST(Act \times WFF(\Sigma)) \rightarrow Prop \times \mathcal{P}(X)$$

$$\begin{array}{lcl} \alpha_2(\epsilon) & = & (\mathsf{true}, \{\}) \\ \alpha_2((\alpha, \phi).t) & = & (\alpha_1(\phi)_1 \cup \alpha_2(t)_1, \alpha_1(\phi)_2 \cup \alpha_2(t)_2 \cup Vars(\alpha)) \\ \alpha_2(t_1 + t_2) & = & (\alpha_2(t_1)_1 \cup \alpha_2(t_2)_1, \alpha_2(t_1)_2 \cup \alpha_2(t_2)_2) \\ \alpha_2(t_1 \parallel t_2) & = & (\alpha_2(t_1)_1 \cup \alpha_2(t_2)_1, \alpha_2(t_1)_2 \cup \alpha_2(t_2)_2) \\ \alpha_2(t \setminus R) & = & \alpha_2(t) \end{array}$$

Once again, we can define an associated concretisation map:

$$\gamma_2(P) = \bigcup\{S \mid A_2(S) \leq P\}$$

where A_2 is α_2 lifted to sets.

Recall that the interpretation of an operator, $op^\sharp$, is *safe* if:

$$op^\sharp \sqsupseteq \alpha \circ op \circ \gamma$$

for suitable α and γ where op is the standard operator. It is fairly routine to verify:

Proposition 2 *The abstract combinators are safe.*

Cousot and Cousot [1] have shown that the safety of the abstract interpretation follows from this local correctness condition.

4.3 An example

The abstract denotation of a program is a conjunction of implications in which an implication $a \leftarrow x$ means that if a is to be expanded then so must x. In the following it is assumed that variable names are unique across all defining equations[1].

We recall an example from Schreiber's thesis (part of his specification of the Alternating Bit Protocol):

$$
\begin{array}{l}
P_1() = accept(accept_1).P_2(1, accept_1) \\
\\
P_2(p_{2,1}, p_{2,2}) = \\
\qquad \overline{send}(p_{2,1}, p_{2,2}).P_2(p_{2,1}, p_{2,2}) \\
\quad + \; ack(ack_1).P_3(p_{2,1}, p_{2,2}, ack_1) \\
\\
P_3(p_{3,1}, p_{3,2}, p_{3,3}) = \\
\qquad \mathsf{if}\ p_{3,1} = p_{3,3}\ \mathsf{then} \\
\qquad\qquad accept(accept_1).P_2(1 - p_{3,1}, accept_1) \\
\qquad \mathsf{else} \\
\qquad\qquad P_2(p_{3,1}, p_{3,2})
\end{array}
$$

The initial environment maps each process to $\lambda\vec{t}.\lambda p.(\mathsf{true}, \{\})$. Assuming an initial call of P_1, after 4 iterations we get the following proposition:

$$
\begin{array}{l}
\{\ (p_{2,2} \leftarrow accept_1), (p_{2,1} \leftrightarrow p_{3,1}), (p_{2,2} \leftrightarrow p_{3,2}), \\
\quad (send_1 \leftarrow p_{2,1}), (send_2 \leftarrow p_{2,2}), (p_{3,3} \leftarrow ack_1)\ \}
\end{array}
$$

(This formula is a simplification of the actual result; we have eliminated self-dependencies).

Note that the occurrences of $accept_1$ in P_1 and P_3 denoted distinct variables in the concrete semantics but are identified in the abstraction. Note, however, that we are obliged to give these variables the same name otherwise the abstraction doesn't work. To see this, consider placing P_1 in parallel with a process p_4 which sends to the channel *accept*. If *accept* is to be expanded in p_4, then it must be expanded in P_1 and P_3. Choosing different names for the variables would mean such information would be lost.

[1] This assumption means that we find the minimal consistent expansion; conflicting names would lead us to identifying more dependencies than actually exist.

5 Conclusions

We have presented a denotational model for value passing process calculi using a generalisation of synchronisation trees. We have also shown how our model can be used as the foundations of an abstract interpretation of value passing programs in order to infer properties of program variables.

The abstract interpretation we have presented can be shown to provide safe information about dependencies between variables. This was also the objective of Schreiber's flow analysis. Are the two analyses equivalent? This remains an issue for further investigation; for the limited number of examples that we have considered, the two analyses produce the same results. It can be argued that our abstract combinators are the *best* that we can define (in the sense of being the induced operators) and thus, we could do no better (with the particular choice of abstraction maps that we have used).

Acknowledgements

We would like to thank Thomas Jensen and Juarez Muylaert-Filho for their comments on an earlier draft of this paper.

References

[1] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analyis by construction or approximation of fixpoints. In *Proc. of the 4th ACM POPL*, pages 238–252, 1977.

[2] P. Dart. *Dependency Analysis and Query Interfaces for Deductive Databases.* PhD thesis, The University of Melbourne, 1988.

[3] P. Dart. On derived dependencies and connected databases. *Journal of Logic Programming*, 11(2):163–188, 1991.

[4] Kim Marriott, Harald Søndergaard, and Neil D. Jones. Denotational abstract interpretation of logic programs. Technical Report 92/20, Department of Computer Science, The University of Melbourne, 1992. To appear in TOPLAS.

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

[6] Robin Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[7] Marcel Zvi Schreiber. *Value-passing Process Calculi as a Formal Method.* PhD thesis, Department of Computing, Imperial College, 1994.

[8] G. Winskel. Event Structures. In *Advances in Petri Nets 1986, Part II*, volume 255 of *Lecture Notes in Computer Science.* Springer, 1987.

[9] G. Winskel and M. Nielsen. Models for concurrency. In S. Abramsky, D. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*. Oxford University Press, 1995. To appear.

[10] Glynn Winskel. Synchronisation trees. *TCS*, 34:33–82, 1984.

[11] Glynn Winskel. An introduction to event structures. In *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, page 364. REX Workshop, 1988. Springer Verlag LNCS 354.