Process Algebras With Localities

Andrew Hughes

Theory SIG - 20/01/2006

Outline

1. Introduction
    - Localities
    - Location Equivalence
2. Locations From A Concrete Perspective
    - Mobility And The π Calculus
    - Amadio's π_{1l} Calculus
    - Fournet and Gonthier's Distributed Join Calculus
    - Cardelli and Gordon's Ambient Calculus
    - Castagna and Vitek's Seal Calculus
3. Conclusion
    - Future Work
    - Final Thoughts
    - Bibliography

What Do We Mean By Localities?

- A locality acts as a way of representing *distribution*.
- It represents the space where a number of processes and resources exist.
- Localities can be observed or controlled.
- Observation of localities is necessary to implement process *migration* between them.
- A locality can be named, and then used as the target for a communication or the destination of a migrating process.

Localities For Equivalence

- The traditional notion of equivalence associated with CCS is bisimulation.
- Bisimulation distinguishes two processes through observing their communication.
- A bisimulation views a parallel process as equivalent to its non-deterministic interleaving.
- However, they differ as the first involves more than one process operating concurrently.
- Practically, the first could be distributed over multiple hosts.

CCS

- We begin by looking at localities in the context of CCS.
- CCS defines processes in terms of the *actions* they can perform.
- We assume a set of names, $\mathcal{N}$, ranged over by $a, b, \ldots$, and a corresponding set of co-names, $\overline{\mathcal{N}} = \{\overline{a} | a \in \mathcal{N}\}$. $\mathcal{N} \cup \overline{\mathcal{N}}$ gives the set of visible actions, *Act*. Silent or internal actions are represented by τ.
- Similarly, we have a set of process variables, $\mathcal{V}$, ranged over by $P, Q, \ldots$. The grammar of CCS is then defined as follows (we omit recursion and relabelling from this definition for brevity):

Definition

$$P, Q ::= 0 \mid a.P \mid \overline{a}.P \mid P \backslash a \mid P + Q \mid (P|Q)$$

Semantics for CCS

Act	$\dfrac{}{\alpha.E \xrightarrow{\alpha} E}$
Sum1	$\dfrac{E \xrightarrow{\alpha} E'}{E + F \xrightarrow{\alpha} E'}$
Sum2	$\dfrac{F \xrightarrow{\alpha} F'}{E + F \xrightarrow{\alpha} F'}$
Com1	$\dfrac{E \xrightarrow{\alpha} E'}{E \mid F \xrightarrow{\alpha} E' \mid F}$
Com2	$\dfrac{F \xrightarrow{\alpha} F'}{E \mid F \xrightarrow{\alpha} E \mid F'}$
Com3	$\dfrac{E \xrightarrow{a} E', F \xrightarrow{\overline{a}} F'}{E \mid F \xrightarrow{\tau} E' \mid F'}$
Res	$\dfrac{E \xrightarrow{\beta} E'}{E \setminus a \xrightarrow{\beta} E' \setminus a} \beta \notin \{a, \overline{a}\}$

Table: CCS SOS Rules

- E and F are processes from the set of process names, $\mathcal{V}$.
- α and β are any actions from $Act \cup \tau$.

A CCS Data Protocol

- Let's take the simple example of a protocol which sends and receives data.
- Our protocol consists of two processes, the *Sender* and the *Receiver*.
- The two communicate using a channel, *a*. This is restricted, giving $Protocol = (Sender|Receiver)\backslash a$
- The sender is simply defined as $Sender = in.\overline{a}.Sender$.
- Similarly, our receiver is $Receiver = a.\tau.\overline{out}.Receiver$.
- Thus, the usual series of actions is $in.\tau.\tau.\overline{out}$, with the first τ being the synchronization on *a*.

Weak Bisimulation

- A bisimulation is a symmetric binary relation, $\mathcal{R}$, between two processes, P and Q.
- The existence of $P\mathcal{R}Q$ and $P \xrightarrow{a} P'$ implies $\exists Q' : Q \xrightarrow{a} Q' \wedge P'\mathcal{R}Q'$.
- For weak bisimulation, we effectively ignore τ transitions. We consider a series of τ transitions, $\xrightarrow{\tau}\xrightarrow{\tau}\ldots$, to be equivalent to $\xRightarrow{\tau}$ and $\xRightarrow{\tau}\xrightarrow{a}\xRightarrow{\tau}$ to be equivalent to $\xrightarrow{a}$.

The Protocol In A Single Process

- We can consider our protocol at a more abstract level by giving it a specification.
- We define this as $PSpec = in.\overline{out}.PSpec$. This views the protocol as a black box, which just takes an input and returns an output, without considering the internal processing.
- By weak bisimulation, this is equivalent to our previous protocol.
- But, our specification can be implemented on only one process, while our earlier implementation actually uses two.
- Even strong bisimulation sees the two as equivalent, if the single process happens to perform the same number of τ actions i.e. $in.\tau.\tau.\overline{out}.0 \sim (in.\overline{a}.0|a.\overline{out}.0)\backslash a$.

General Bisimulation Problems

- More generally, we can take a process such as $a.0|b.0$, where $b \neq \overline{a}$.
- The equivalent interleaving of this process is thus $a.b.0 + b.a.0$.
- Again, the two are strongly bisimilar, but yet the first runs on two processes, while the first only runs on one.
- If our system is distributed, with our processes actually being on separate hosts, then it may be important for us to distinguish between these two cases.
- Thus, we need a different equivalence to tell the two apart.
- This is how localities originated.

Adding Localities

- We can add an additional piece of syntax to CCS: $l :: P$.
- This specifies that P is located at $l \in Loc$, the set of localities.
- There are two different approaches to providing semantics with this additional syntax. The **static approach** assigns localities beforehand, and they are observed within the transitions. In contrast, the **dynamic approach** generates localities as part of each transition, making each locality an identifier for each non-silent action. This leads to the generation of a *causal path*.
- Here, we will just consider the dynamic approach. Further details on location equivalence, including proofs and details of the static approach, are available in [GA93] and [Cas01].

Transition Semantics for LCCS

Act1 $\dfrac{}{\alpha.E \xrightarrow[l]{\alpha} l :: E}$ *for any* $l \in Loc$	Act2 $\dfrac{E \xrightarrow[u]{\alpha} E'}{l :: E \xrightarrow[lu]{\alpha} l :: E'}$
Sum1 $\dfrac{E \xrightarrow[u]{\alpha} E'}{E + F \xrightarrow[u]{\alpha} E'}$	Sum2 $\dfrac{F \xrightarrow[u]{\alpha} F'}{E + F \xrightarrow[u]{\alpha} F'}$
Com1 $\dfrac{E \xrightarrow[u]{\alpha} E'}{E \mid F \xrightarrow[u]{\alpha} E' \mid F}$	Com2 $\dfrac{F \xrightarrow[u]{\alpha} F'}{E \mid F \xrightarrow[u]{\alpha} E \mid F'}$
Res $\dfrac{E \xrightarrow[u]{\beta} E'}{E \setminus a \xrightarrow[u]{\beta} E' \setminus a}$ $\beta \notin \{a, \overline{a}\}$	

Table: LCCS SOS Rules

- u is any location.
- Note that τ transitions are not assigned locations.

Location Equivalence

- We can now define an equivalence based on localities.
- A relation, $R \subseteq LCCS \times LCCS$ is called a *dynamic location bisimulation* (*dlb*) iff for all $(p, q) \in R$ and for all $a \in Act, u \in Loc$:

1. $P \overset{a}{\underset{u}{\Rightarrow}} P' \implies \exists Q'$ *such that* $Q \overset{a}{\underset{u}{\Rightarrow}} Q'$ *and* $(P', Q') \in R$
2. $Q \overset{a}{\underset{u}{\Rightarrow}} Q' \implies \exists P'$ *such that* $P \overset{a}{\underset{u}{\Rightarrow}} P'$ *and* $(P', Q') \in R$
3. $P \overset{\tau}{\Rightarrow} P' \implies \exists Q'$ *such that* $Q \overset{\tau}{\Rightarrow} Q'$ *and* $(P', Q') \in R$
4. $Q \overset{\tau}{\Rightarrow} Q' \implies \exists P'$ *such that* $P \overset{\tau}{\Rightarrow} P'$ *and* $(P', Q') \in R$

- The largest *dlb* is called *dynamic location equivalence*.

Back To The Protocol

- With this equivalence, we can distinguish between *PSpec* and *Protocol*.
- With LCCS, *Protocol* has the sequence of transitions $\xrightarrow[l]{in}\xrightarrow{tau}\xrightarrow{tau}\xrightarrow[k]{out}$.
- *PSpec* has the transition sequence $\xrightarrow[l]{in}\xrightarrow[lk]{out}$.
- Thus, *Protocol* ends up as $(l :: Sender | k :: Receiver) \setminus a$.
- *PSpec* ends up as $l :: k :: PSpec$.
- The sequences clearly differ. *PSpec* has a history of locations, resulting from the two transitions taking place on the same process. However, the transitions in *Protocol* take place on separate processes, leading to two separate locations. Note that it is not the identity, but the distribution of these localities that is important.

A Chance In Perspective

- So far we have looked at localities from the perspective of enriching existing equivalence theories.
- The localities in this context have been fairly *abstract*, in that they exist solely as a way to distinguish the distribution of processes.
- Calculi with a concrete notion of localities allow the localities to be observable and have identities.
- Most notably, we can use localities as a means to provide a different form of *mobility*.

What Is Mobility?

- Mobility is probably most well-known from the π calculus.
- However, mobility is the π calculus is not so much to do with processes, as it is to do with *scope*.
- We can't really move processes in the π calculus because they have no distribution i.e. we don't know where they are to start with!
- Migration of processes from one place to another is only possible if we add a notion of location to the calculus. Probably the simplest way to do this is as we have already seen; by assigning localities to the processes.

The Mini π Calculus

- The π calculus has provided a useful basis to several distributed calculi. It is basically a value-passing variant of CCS, with the generalisation of both variables and channels into a common set of **pure names**.
- The mini-π calculus was introduced by Milner in [Mil92] as a subset of the full π calculus. Notably, it doesn't include either the match or summation operators, or the agent notation.

Definition

$$P, Q ::= 0 \mid x(u).P \mid \overline{x}\langle u\rangle.P \mid (\nu x)P \mid (P|Q) \mid !P$$

- Again, P and Q are processes. x and u are both names, as there is no distinction between variables and channels.

Variants Of The π Calculus

- The asynchronous variant is also commonly used. This is derived by simply replacing $\overline{x}\langle u\rangle.P$ with $\overline{x}\langle u\rangle$, making output non-blocking.
- Replication ($!P$) may also be replaced by recursion.
- A polyadic variant can also be created by generalising the input and output prefixes to use vectors ($x(\vec{y})$ and $\overline{x}\langle\vec{y}\rangle$).

Origins Of The Calculus

- The π_{1l} calculus originated as the π_l calculus in a paper[AP94] by Amadio and Prasad to give failure semantics to the language, Facile. This added a flat notion of locations to the synchronous polyadic π calculus.
- The version we will consider was published in a later paper [Ama97] by Amadio, and is instead based on the *asynchronous* variant of the full calculus.
- It in fact builds on the π_1 calculus, which is an asynchronous typed variant satisfying the *unique receiver* property.
- With this property, each channel has at most one receiver. The result is that the destination of an output is pre-determined. This property is enforced by the type system of the calculus.

Extensions Within The Calculus

- The calculus is concerned primarily with the detection of failure.
- Thus, it adds syntax to model failure and its detection. Failure is associated with a particular location, so syntax is also added to represent these locations, as we saw earlier.
- $l :: P$ represents a process, P, running at the location, l. Note that we continue with our previous notation rather than using that given in the paper.
- Outputs are generalised into a larger category of *messages*, which includes additional primitives:
  - $stop(l)$, which stops a location, l.
  - $spawn(l, P)$, which spawns a process P at l.
  - $ping(l, b_1, b_2)$, which checks that the location l is running, and sends a message on either b_1 or b_2, depending on the result.

Main Features

- *Objective* migration – whichever process contains $spawn(l, P)$ causes the process P to move.
- Migration also occurs via message passing.
- Each locality has a *locality-process*, which records the status of the location and handles *spawn*, *ping* and *stop* requests.
- Global communication, but more elegant due to asynchrony and the unique receiver property.
- The π_{1l} calculus can encode the π_1 calculus, which in turn can encode the π calculus.

The Join Calculus

- The join calculus [FG96] is another variant of the asynchronous π calculus. The differences lie in the *receptors*.
- They differ from those in the π calculus in that:
  1. Localisation is enforced in the syntax and scoping discipline. The inputs are defined in the same statement as the output they connect to.
  2. Channel receptors are permanently defined, and are not on a one-shot system like in the π calculus. Thus, a join calculus input is akin to a replicated input (e.g. $!x(y).P$) in the π calculus.
  3. Every channel must be statically defined, unlike in the π calculus which allows $(\nu z)z(y).(x(u).P|y(v).Q)|\overline{z}\langle x\rangle$. Names are bound by their definition, so receptors can't be renamed and two different receptors can never be equated.

The π and Join Calculi

- The changes in the join calculus make the calculus easier to implement in a distributed way.
- In the π calculus, we can define:

Definition

$$x(y).P|x(z).Q|\overline{x}\langle u\rangle$$

- If the two receptors, $x(y).P$ and $x(z).Q$ are far apart, this runs into a *distributed consensus problem*, as a decision has to be made over which process takes the output.
- The join calculus avoids this by changing the syntax to:

Definition

$$\mathbf{def}\ (x\langle y\rangle \triangleright P) \wedge (x\langle z\rangle \triangleright Q)\ in\ x\langle a\rangle$$

Join Calculus Syntax Changes

- In fact, the syntax of the join calculus means that the above is actually the analogue of the following π calculus definition:

Definition

$$(\nu x)(!x(y).P|!x(z).Q|\overline{x}\langle u\rangle)$$

- This makes join calculus receptors localised, permanently available and statically defined.
- The syntax overloads the same notation for input and output, as the two are differentiated by their position in the syntax.

Join Patterns

- Asynchronous messaging means that only a single simple message can be transmitted. To allow synchronization, the join calculus includes *join patterns* to define groups of messages. Names in a pattern must be distinct, but names in different conjuncts need not. Simultaneous substitution takes place as a result, and non-determinism may occur.

Definition

$$\mathbf{def}\ (x\langle y\rangle | t\langle u\rangle \triangleright P) \wedge (x\langle z\rangle | t\langle v\rangle \triangleright Q)\ in\ x\langle a\rangle | t\langle c\rangle | x\langle b\rangle$$

Reduction in the join calculus

- This process is generalised to form a reduction rule. This forms the crux of the semantics for the calculus.

Definition

$$\textbf{def } (D \wedge J \triangleright P) \textit{ in } J\sigma | Q \rightarrow \textbf{def } (D \wedge J \triangleright P) \textit{ in } P\sigma | Q$$

- In addition, standard contextual rules and a structural congruence $\equiv$ are defined. The latter allows **def** to be pushed in front of a term.
- The semantics were originally defined using a *Chemical Abstract Machine* (CHAM) proposed by Berry and Boudol [GG92].

The Distributed Variant

- The distributed variant [CF96] adds locations and primitives for migration.
- A *located declaration* is added of the form $l[D : P]$.
- This defines the input channels *located* at l.
- Again, the locality is scoped over its **def** rule and the declaration is unique and global.
- However, localities are unique within the rule, unlike channels.
- Receptors must be defined i.e. T is not a valid definition.
- Localities can be nested, giving a hierarchical structure.

Definition

$$\textbf{def } a[x\langle y\rangle \triangleright P : Q \wedge x\langle z\rangle \triangleright Q : R)\ in\ S$$

Migration

- The new process construct, $go\langle l, k\rangle$ allows the migration of processes.
- Migration is *subjective*, unlike in Amadio's calculus. The *go* construct moves the locality in which the executing process resides to become a sublocation of l.
- Upon termination of the migration, a null message, $k\langle\rangle$ is emitted.
- The moving locality, m, must not be a superlocality of l, as its entire subtree is also moved.
- Structuring is also important in failure, as all sublocations fail too. Failure detection is provided by additional *halt* and *detect* messages.

Expressivity of the Join Calculus

- The distributed join calculus is equivalent to the join calculus where *circular migration* does not occur. The asynchronous π calculus can be encoded in the join calculus, and vice versa.

A Different Perspective

- The ambient calculus [CA98] emphasises mobility over communication, whereas the reverse could be said of the π calculus.
- The mobility primitives are sufficient for the full expressiveness of the calculus, and the communication primitives are encoded using these.
- *Ambients* are named bounded areas with a collection of processes and subambients.
- As with the join calculus, migration is subjective and moves the entire subtree. However, processes can also dissolve boundaries using the *open* primitive.
- Processes within an ambient communicate using names, *capabilities* or sequences of these by emitting into the local ether.

Ambient Movement

- Ambients can only enter sibling ambients and exit parent ambients.
- Only ambients within the same parent can be opened.
- This gives *proximity mobility*, which is appropriate in the context of *hierarchical administrative domains*, which the calculus was designed to model.
- The capabilities model *authorisation*, and administrate ambient movement.
- Ambients are written as $n[P]$ where n is its name and P its contents. The core mobility grammar is:

Definition

$$P, Q ::= 0 \mid M.P \mid P|Q \mid (\nu n)P \mid !P \mid n[P]$$

Mobility Constructs

- Reductions in the ambient calculus take place equally outside as well as inside ambients, even when the surrounding ambient is moving.
- i.e. $P \rightarrow Q \Rightarrow n[P] \rightarrow n[Q]$
- The mobility constructs (M above) are:
  1. *in n.P* which moves the surrounding ambient inside n e.g. $n[\mathbf{in}\ m.P|Q]|m[R] \rightarrow m[n[P|Q]|R]$
  2. *out n.P* moves the surrounding ambient out of the parent ambient n e.g. $m[n[\mathbf{out}\ m.P|Q]|R] \rightarrow n[P|Q]|m[R]$
  3. *open n.P* opens the ambient n e.g. $\mathbf{open}\ m.P|m[Q] \rightarrow P|Q$

Further Points On The Calculus

- The same name may coexist both at the same level and at different levels of the hierarchy.
- One is chosen non-deterministically.
- Empty ambients are still observable.
- Same-named ambients are distinct.
- Ambients resemble the named locations we saw earlier, but are more like *mobile agents*.
- The calculus can encode the asynchronous π calculi and some λ calculus. A representation of a Turing machine is also given as an example in the paper.

Combining The Two

- The Seal calculus [VC99] may be described as a polyadic synchronous variant of the π calculus.
- But, it follows many ideas seen in the Ambient calculus when it comes to modelling networks and security concerns.
- The calculus introduces a new type of name, the *seal*. A seal ($n[P]$) is a process and can encapsulate other processes, again giving us a hierarchical structure.
- Primitive communication is restricted to *local* communication and *linear proximity* communication. Further, more distant communication must be routed.

Channels In The Seal Calculus

- Channels are tagged with a notation that specifies where they belong.
- The tags are defined by the following grammar:

Definition

$$\eta ::= \star \mid \uparrow \mid n$$

- $\star$ refers to the **current seal**
- $\uparrow$ refers to the **parent seal**
- n is the name of a child seal.

Communication

- Channel communication is much the same as in the π calculus.

Definition

$$a ::= \overline{x}^{\eta}(\vec{y}) \mid x^{\eta}(\vec{y})$$

- **Local communication** takes place between two channels tagged with $\star$.
- **Upward or downward communication** takes place between one channel tagged with $\star$ and another tagged with $\uparrow$ or n.

Portals

- Non-local communication is influenced by security.
- For seal A to communicate with channel x in seal B, B must first open a *portal* to allow this.
- A *portal* forms a means of *linear access permission* for a channel, which may only be used once.
- This is represented by the notation $open_s x.P$ which opens a portal for seal s and then continues as P.

Example

$$n[\overline{x}^{\uparrow}(\vec{z}).P]|x^{\star}(\vec{y}).Q|open_n\overline{x}.0 \rightarrow n[P]|Q\{\vec{z}/\vec{y}\}|0$$

Seal Mobility

- Seals may be transmitted over channels, and this forms the mobility within the seal calculus.
- *a* is extended with two prefixing actions for transmitting seals.

Definition

$$a ::= \overline{x}^{\eta}\{y\} \mid x^{\eta}\{\vec{y}\}$$

- Note that only a single seal name can be output. Copies of the same seal are placed in the input vector.

The Effects Of Seal Movement

- A seal is moved by a process contained in the parent. Thus, mobility is objective, unlike in the ambient calculus.
- *Renaming* and *duplication* may both take place during the movement of seals.
- If $P = \overline{x}^{\uparrow}\{y\}.P'$ and $R = x^{\star}\{z\}.R'$, then:

Example

$$R|n[P|m[Q]|y[S]]|open_n\overline{x}.0 \rightarrow R'|z[S]|n[P\prime|m[Q]]$$

- Note that the seal always moves from the sender to the receptor seal, so $\overline{x}^{\star}$ in P and x^{n} in R would also have worked.
- Also, this is *spawn* in disguise; a new locality is created through renaming with the contents of the old one.

Comparing The Seal Calculus With The Ambient Calculus

- The ambient calculus is one of the sources of inspiration for the seal calculus.
- However, the seal calculus demonstrates *objective* mobility and an emphasis on communication, in contrast with the ambient calculus.
- Environmental control is preferred over capabilities.
- There is no equivalent of the dangerous *open* construct.
- Both satisfy the *perfect firewall equation*: $(\nu x)x[P] = 0$ i.e. a process can be completely isolated.

Using Localities

- Localities seem to provide a more practical form of mobility than that demonstrated by the π calculus.
- In addition to giving migration, localities also allow us to know where a process 'is'. With this knowledge, we can:
    1. Consider process distribution.
    2. Observe failure.
    3. Represent hierarchical structures and other physical notions such as hosts on a network.

Combining Localities With Time

- Recall the Cashew-Nuts calculus...
- This has an implicit notion of hierarchy in the form of clock hiding.
- Could localities be combined with this notion to make this explicit?
- This would also allow these hierarchies to be observed and migrated.
- Nomadic Nuts... ;)

In Conclusion. . .

- Localities can be used in a variety of ways.
- We have seen:
  1. A form of bisimulation, using localities to represent distribution.
  2. A concrete notion of locality added to the π calculus to allow migration and failure detection.
  3. A hierarchy of localities in the join, ambient and seal calculi, which give structure to the processes represented.
- The mailing list (`theory@dcs.shef.ac.uk`) and wiki are available for further discussion.
- Thanks for listening.

`http://www.dcs.shef.ac.uk/wiki/bin/view/TheorySIG`

References

R. Amadio.
An asynchronous model of locality, failure and process mobility.
In *COORDINATION 97*, number 1282 in LNCS, 1997.

R. Amadio and S. Prasad.
Localities and failures.
In *FST-TCS 94*, number 880 in LNCS, pages 205–216, 1994.

L. Cardelli and A.D.Gordon.
Mobile ambients.
In *FoSSaCS 98*, number 1378 in LNCS, 1998.

References

- I. Castellani.
  Process Algebras With Localities, pages 945–1046.
  North-Holland, 2001.
- C. Fournet, G. Gonthier, J.-J. Lévy et al.
  A calculus of mobile agents.
  In *CONCUR 96*, number 1119 in LNCS, 1996.
- C. Fournet and G. Gonthier.
  The reflexive chemical abstract machine and the join-calculus.
  In *POPL 96*, pages 372–385, 1996.

References

M.Hennessy G.Boudol, I. Castellani and A.Kiehn.
Observing localities.
Theoretical Computer Science, 114:31–61, 1993.

G.Berry and G.Boudol.
The chemical abstract machine.
Theoretical Computer Science, 96:217–248, 1992.

R. Milner.
Functions as processes.
Mathematical Structures in Computer Science, 2(2):119–141, 1992.

References

J. Vitek and G. Castagna.
Seal: A framework for secure mobile computations.
In D. Tsichritzis, editor, *Workshop on Internet Programming Languages*, 1999.