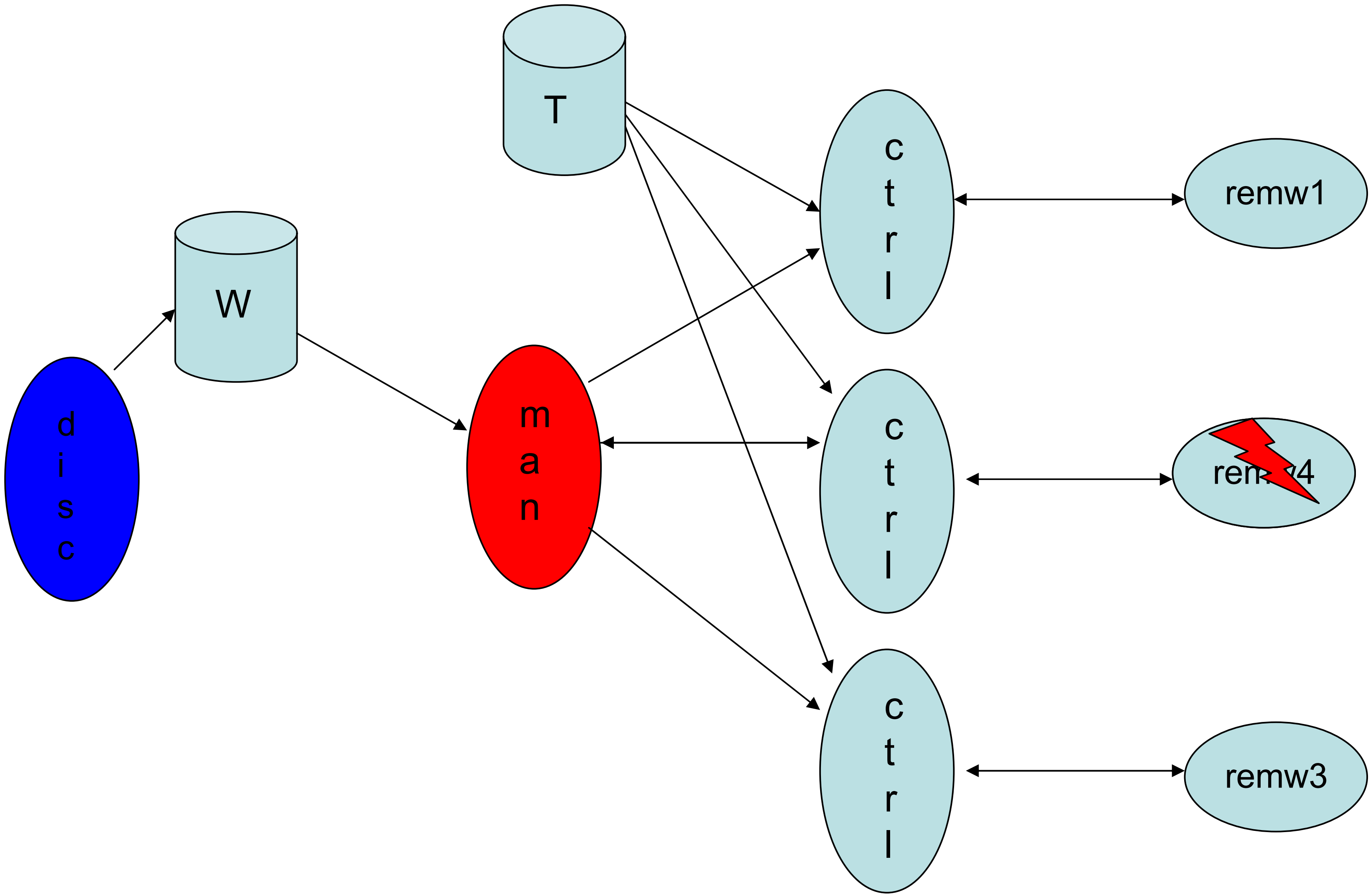
Using Orc to aid Design and Analysis of Dynamic Distributed Systems

..and in particular with respect to non-functional properties

“Currently, most of the effort is concentrated on the ends of the spectrum, which are far from the designer's viewpoint. For example, BPEL is a recognized standard for orchestration of Web Services, but it is designed for machine processing ….

At the other extreme, π-calculus is a well-recognized formal tool for reasoning about distributed programs, but it comes with a heavyweight formal framework typically outside the interest and experience of system designers.”

Muskel



system

```
system(pgm, tasks, contract, G, t) =
        taskpool.add(tasks)
      | discovery(G, pgm, t)
      | manager(pgm, contract, t)
```

discovery

```
discovery(G, pgm, t) =
    (|g ∊ G  ( if (remw ≠ false) >> rworkerpool.add(remw)
                  where remw :∊
                      ( g.can_execute(pgm) | Rtimer(t) >> let(false) )
               )
     )   >> discovery(G, pgm, t)
```

manager

manager(pgm, contract, t) =

|i : 1 ≤ i ≤ contract : (rworkerpool.get > remw >

ctrlthread_i(pgm, remw, t))

| monitor

ctrlthread

```
ctrlthreadi(pgm, remw, t) =
  taskpool.get > tk >
        (   if valid >> resultpool.add(r) >> ctrlthreadi(pgm, remw, t)
          | if ¬valid >> (  taskpool.add(tk)
                           | alarm.put(i)  >> ci.get > w >
                                                    ctrlthreadi(pgm,w, t)
                           )
        )
   where
     (valid, r)  :∈ (  remw(pgm, tk) > r > let(true, r)
                     | Rtimer(t) >> let(false, 0)
                     )
```

monitor

```
monitor =
    alarm.get > i > rworkerpool.get > remw > ci.put(remw) >>
                    monitor
```

Analysis

- Manager is responsible for recruitment and supply of remote workers to control threads, initially and after remote worker failure.
- Manager represents a single point of failure.
- Aim: to make each control thread responsible for its own remote worker supply, thus removing single point of failure.

Strategy

- Examine traces of site calls made by processes
- Identify management related activity
- Try to identify where/how functionality can be dispersed to disperse this management activity

Strategy

- Typically communication occurs when a process, A, generates a value, x, and communicates it to B.
- Identify occurrences of this pattern and consider if generation of the item could be shifted to B and the communication removed, with the “receive” in B being replaced by the actions leading to x’s generation.

Derivation

A : . . . a1, a2, a3, send(x), a4, a5, . . .

B : . . . b1, b2, b3, receive(i), b4, b5, . . .

Assume that a2, a3 (which, in general, may not be contiguous) are responsible for generation of x, and *it is reasonable to transfer this functionality* to B. Then the above can be replaced by:

A : . . . a1, a4, a5, . . .

B : . . . b1, b2, b3, a2, a3, (b4, b5, . . .)[i/x]

Derivation

In control thread:

alarm.put(i) >> c_i.get > w > $ctrlthread_i$(pgm,w, t) . . .

In monitor:

alarm.get > i > rworkerpool.get > remw > c_i.put(remw)

Move remote worker (remw) “generation” to the control thread.

Derivation

In control thread:

alarm.put(i) >> rworkerpool.get > remw >

$ctrlthread_i$(pgm, remw,t) . . .

In monitor:

alarm.get > i > . . .

Derivation

In control thread:

alarm.put(i) >> rworkerpool.get > remw >
ctrlthread_i(pgm, remw,t) . . .

In monitor:

alarm.get > i > . . .

Revised muskel spec.

systemD(pgm, tasks, contract, G, t) =

taskpool.add(tasks)

$|i : 1 \leq i \leq$ contract : ctrlthread$_i$(pgm, t,G)

ctrlthread$_i$(pgm, t,G) =

discover(G, pgm) > remw > ctrlprocess(pgm, remw, t,G)

discover(G, pgm) =

let(remw) where remw $:\in (|_{g \in G}$ g.can execute(pgm))

Revised muskel spec.

```
ctrlprocess(pgm, remw, t,G) =
  taskpool.get > tk >
    (   if valid >> resultpool.add(r) >> ctrlprocess(pgm, remw, t,G)
      | if ¬valid >>  taskpool.add(tk)
                         | discover(G, pgm) > w > ctrlprocess(pgm,w, t,G)
    )
     where (valid, r) :∈
               (   remw(pgm, tk) > r > let(true, r)
                 | Rtimer(t) >> let(false, 0)
               )
```

Key ideas

- Orc model allows the essence of the structure to be seen devoid of implementation detail.
- This model may be used to analyse the system and investigate its properties.
- For example:
  - Original muskel spec: “core processing” and discovery are composed using the parallel operator.
  - Modified spec: “core processing and discovery are composed using “>>”.
  - This suggests a price to pay in efficiency.

Key ideas

- The style emphasises a semi-formal approach. That is, using a formal notation (with a well-defined) semantics but not providing formal proofs and drawing on insight and experience to justify steps.

- Orc is seen to be appropriate for developing/analysing systems such as muskel:
  - Small readable syntax.
  - Constructs suitable for describing typical activities such as parallel search, time-out, etc.
  - Site abstraction provides clear separation between core functionality and management.

Reference

Marco Aldinucci, Marco Danelutto, Peter Kilpatrick.

Management in distributed systems: a semi-formal approach.

In *A.-M. Kermarrec, L. Bouge and T. Priol, editors, Proc. of 13th Intl. Euro-Par 2007, LNCS,* Rennes, France, Aug. 2007.

Orc metadata

- Enrich Orc with metadata to describe non-functional properties such as deployment information.

- Introduce a new dimension for reasoning about the orchestration of a distributed system by allowing a narrowing of the focus from the very general case.

Orc metadata

- An Orc program is a set of Orc definitions followed by an Orc goal expression. The goal expression is the expression to be evaluated when executing the program.

- Assume $S = \{s_1, . . . , s_n\}$ is the set of sites used in the program (not including predefined sites).

- $E = \{e_0, . . . , e_e\}$ is the set that includes the goal expression (e_0) and all the "head" expressions appearing in the left hand sides of Orc definitions.

Orc metadata

- $M = \{\mu_1, ... , \mu_n\}$

  where $\mu_i = <t_j ,md_k>$

  with $t_j \in S \cup E$ and $md_k = f(p_1, ... , p_{nk})$.

- f is a generic "functor" (represented by an identifier) and p_i are generic "parameters" (variables, ground values, etc.).

Example: site placement

- Suppose one wishes to reason about Orc program site "placement", i.e. about information concerning the relative positioning of Orc sites with respect to a given set of physical resources potentially able to host one or more Orc sites.

- Let $R = \{r_1, . . . , r_r\}$ be the set of available physical resources.

- Then, given a program with $S = \{siteA, siteB\}$ we can consider adding to the program metadata such as

  $$M = \{<siteA, loc(r_1)>, <siteB, loc(r_2)>\}$$

  modelling the situation where siteA and siteB are placed on distinct processing resources.

Example: site placement

- Define also the auxiliary function

  $$location(x) : S \times E \rightarrow R$$

  as the function returning the location of a site/expression

- the cost of a communication with respect to the placement of the sites involved can be characterized by distinguishing cases:

  $$k_{Comm} = \begin{cases} k_{nonloc} & \text{if } location(s_1) \neq location(s_2) \\ k_{loc} & \text{otherwise} \end{cases}$$

  where s_1 and s_2 are the source and destination sites of the communication, respectively and, typically, $k_{nonloc} >> kl_{oc}$.

Example: security

- Suppose "secure" and "insecure" site locations are to be represented.

- Add to the metadata tuples such as $<s_i, trusted()>$ or $<s_i, untrusted()>$.

- $$k_{SecComm} = \begin{cases} k_{InSecComm} & \text{if } <s1, untrusted()> \in M \vee <s2, untrusted()> \in M \\ k_{Comm} & \text{otherwise} \end{cases}$$

Metadata generation: placement metadata

- *Completely distributed strategy:* it is assumed that each time a new site in the Orc program is encountered, the site is "allocated" on a location that is distinct from the locations already used.

- *Conservative strategy*: new sites are allocated in the same location as their parent (w.r.t. the syntactic structure of the Orc program), unless the user/programmer specifies something different in the provided metadata.

- Then, for example, an Orc spec. can be analysed w.r.t. communication cost based on metadata.

Reference

Marco Aldinucci, Marco Danelutto, Peter Kilpatrick.

Adding Matadata to Orc to Support Reasoning aboutGrid Programs.

In T Priol, M Vanneschi, *Proc. of CoreGRID Symposium 2007,* Rennes, France, Aug. 2007.

Other Work

- Using Orc to model Grid Programming:

  A. Stewart, J. Gabarro, M. Clint, T.Harmer, P. Kilpatrick, R.Perrott. Managing grid computations: an ORC-based approach. In: *M Guo et al (Eds) Proc. International Symposium on Parallel and Distributed Processing and Applications (ISPA 06)*, Sorrento. LNCS 4330. Springer. pp 278-291. 4-6 December 2006.

- Probabilistic Reasoning about the Reliability of Grid applications using Orc models:

  A. Stewart, M. Clint, T.Harmer, P. Kilpatrick, R. Perrott, J. Gabarro. Estimating the reliability of Web and Grid Orchestrations. In: S Gorlatch, M Brubak, T Priol (Eds.) Proceedings of the CoreGRID Integration Workshop, Krakow. pp. 141-152. ISBN: 83-915141-6-1. 19-20 October 2006.

Other Work

- Orc -> Partial order of events -> Probabilistic analysis using *TOrQuE tool.*

  Sidney Rosario Albert Benveniste Stefan Haar Claude Jard.

  Probabilistic QoS and soft contracts for transaction based Web services orchestrations.

  In *Proc. of IEEE Int. Conf. on Web Services (ICWS),* July 9-13, Salt Lake City, 2007.

- Using Game theory to reason about reliability of Grid systems.

  Joaquim Gabarró, Alina García, Maurice Clint, Peter Kilpatrick, Alan Stewart. Bounded Site Failures: An Approach to Unreliable Grid Environments. *CoreGRID Workshop on Grid Programming Model, Grid and P2P Systems Architecture and Grid Systems, Tools and Environments.* Heraklion - Crete, Greece, June 12-13, 2007.