TOWARDS HIERARCHICAL MANAGEMENT OF AUTONOMIC COMPONENTS: A CASE STUDY

EUROMICRO PDP 2009, 18-21 FEB, WEIMAR, GERMANY

MARCO ALDINUCCI
COMPUTER SCIENCE DEPT. - UNIVERSITY OF TORINO - ITALY

MARCO DANELUTTO
COMPUTER SCIENCE DEPT. - UNIVERSITY OF PISA - ITALY

PETER KILPATRICK
COMPUTER SCIENCE DEPT. - QUEEN'S UNIVERSITY BELFAST - U.K.

Outline

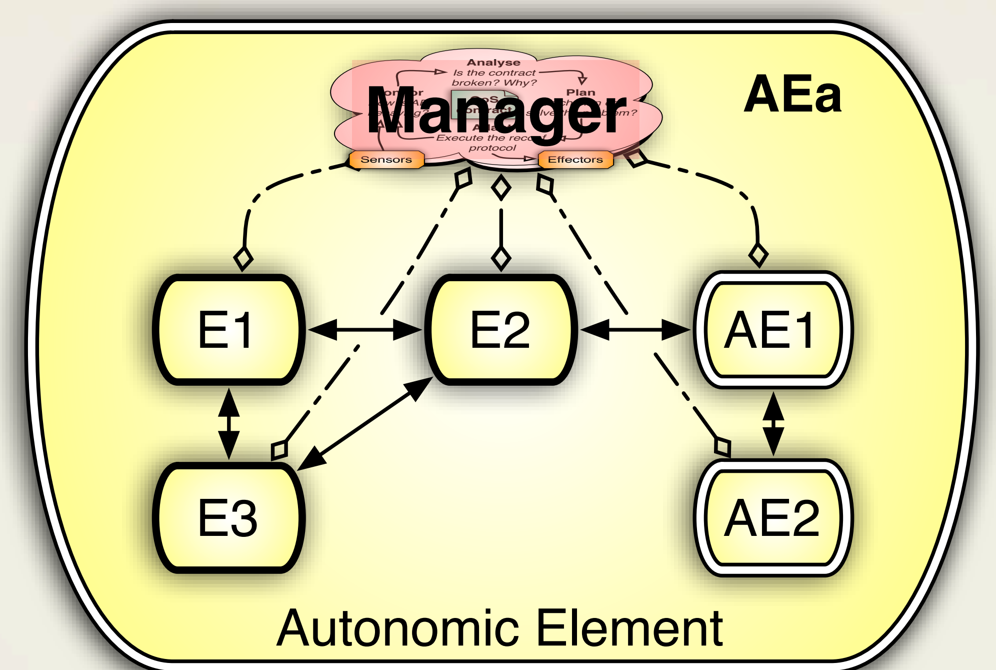
* A semi-formal framework for autonomic components
  * rigorously defining autonomic cycle
  * rigorously defining managers behaviour
* Behavioural skeletons and component hierarchy
  * decoupling management from business code, coupling behaviour with skeletons/patterns
  * easing autonomic applications design by way of automatic manager generation
* Demo*nstrating all above

ON INTRODUCING A SEMI-FORMAL FRAMEWORK FOR AUTONOMIC COMPONENTS

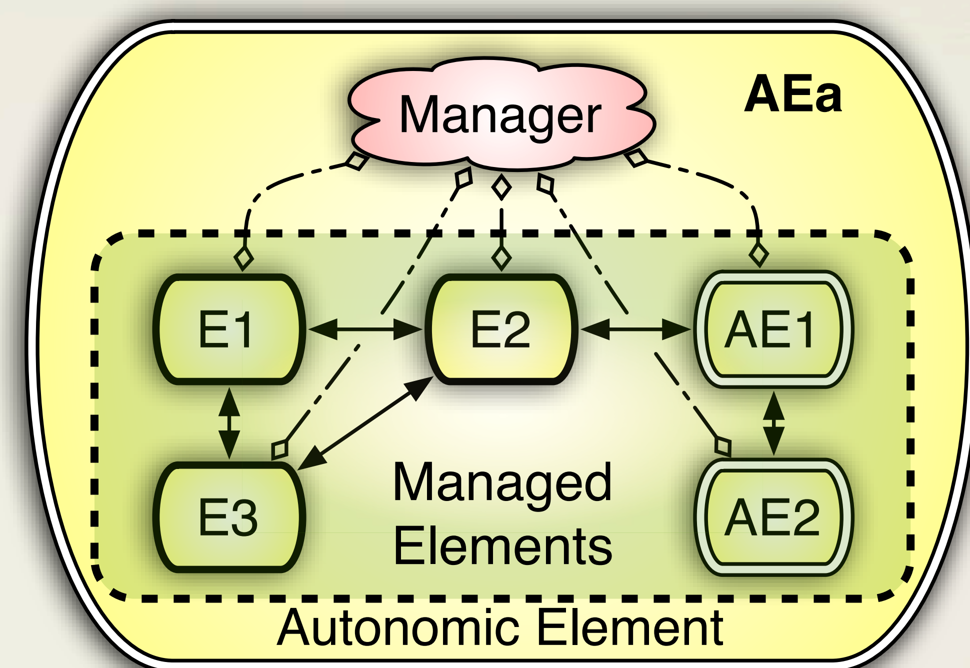
Autonomic Components

* CoreGRID Grid Component Model (GCM)
  * recently standardised by ETSI (July 2008)
  * not only grid, but also distributed and multi/may core
* GCM
  * use-provide ports, RPC, events, streams
  * broadcast, multicast, unicast, gather ports
  * hierarchic: components can be nested, derived from Fractal component model, which don't cover concurrency/parallelism
  * prototypal implementation (GCM/Proactive)
    * autonomic features designed and developed in GridCOMP
    * already used for real world application, see IBM, GridSystem, ATOS, ...

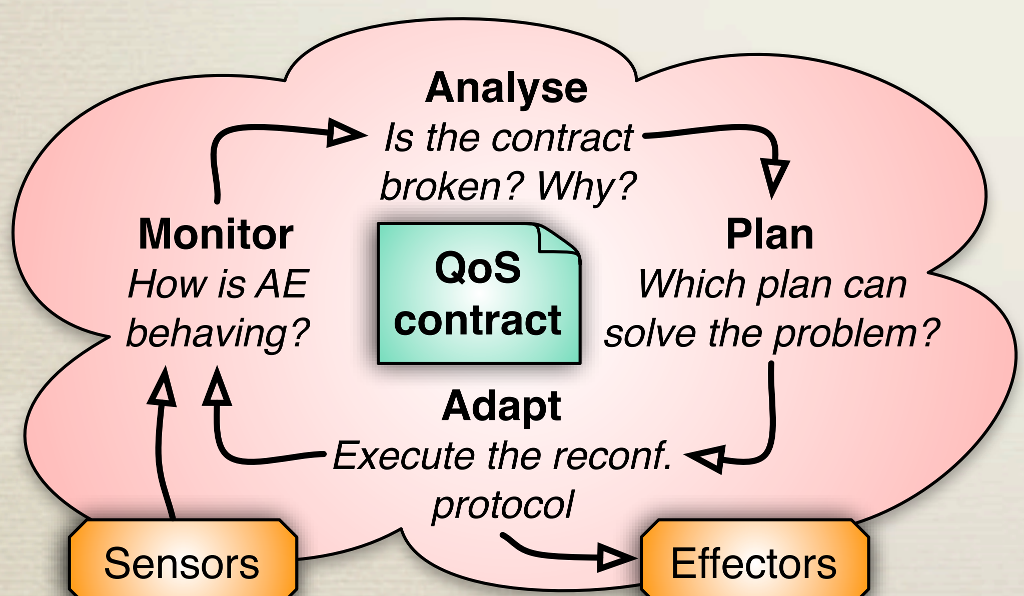
Autonomic Components



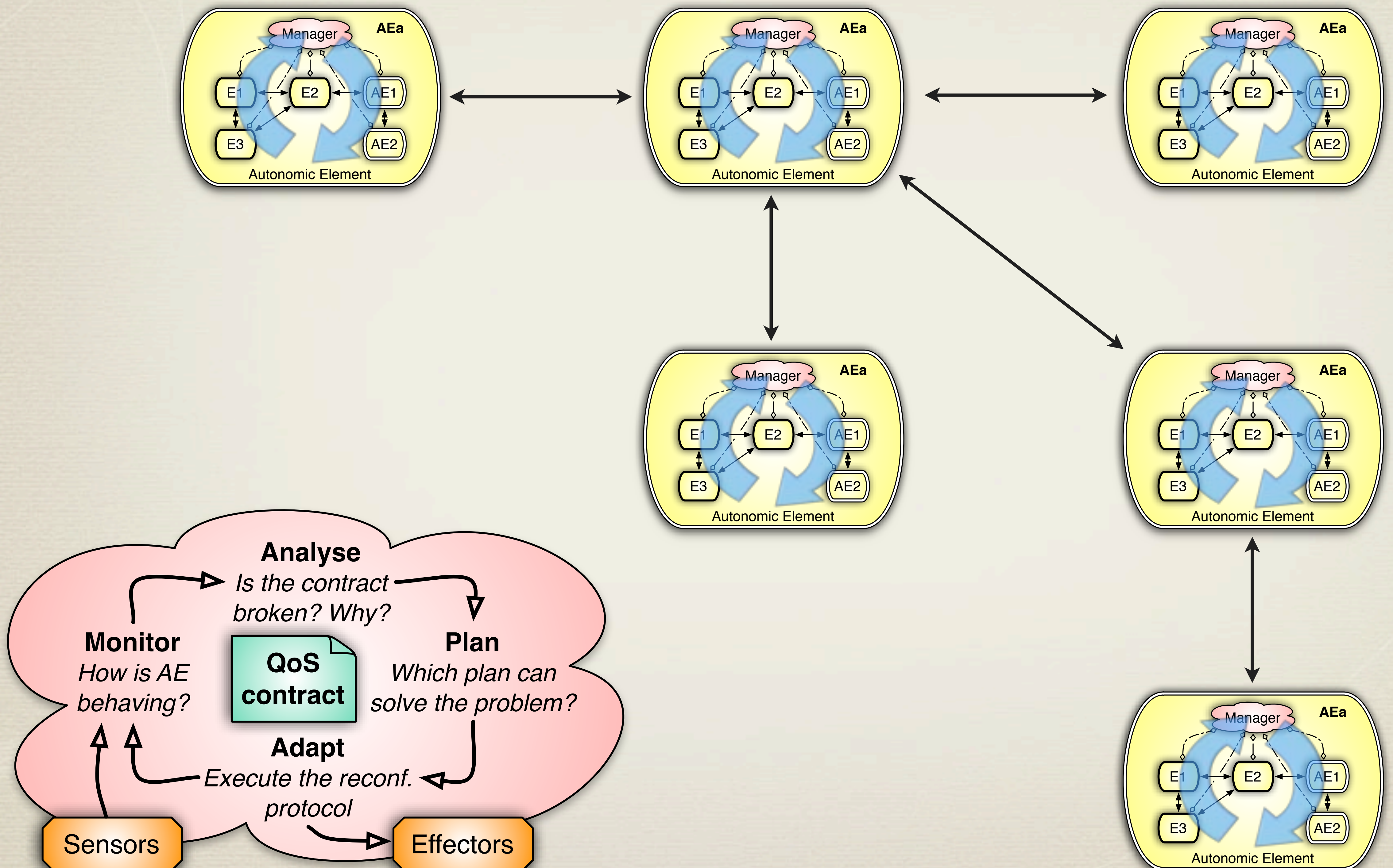
Autonomic Components



Manager life cycle



Autonomic Components



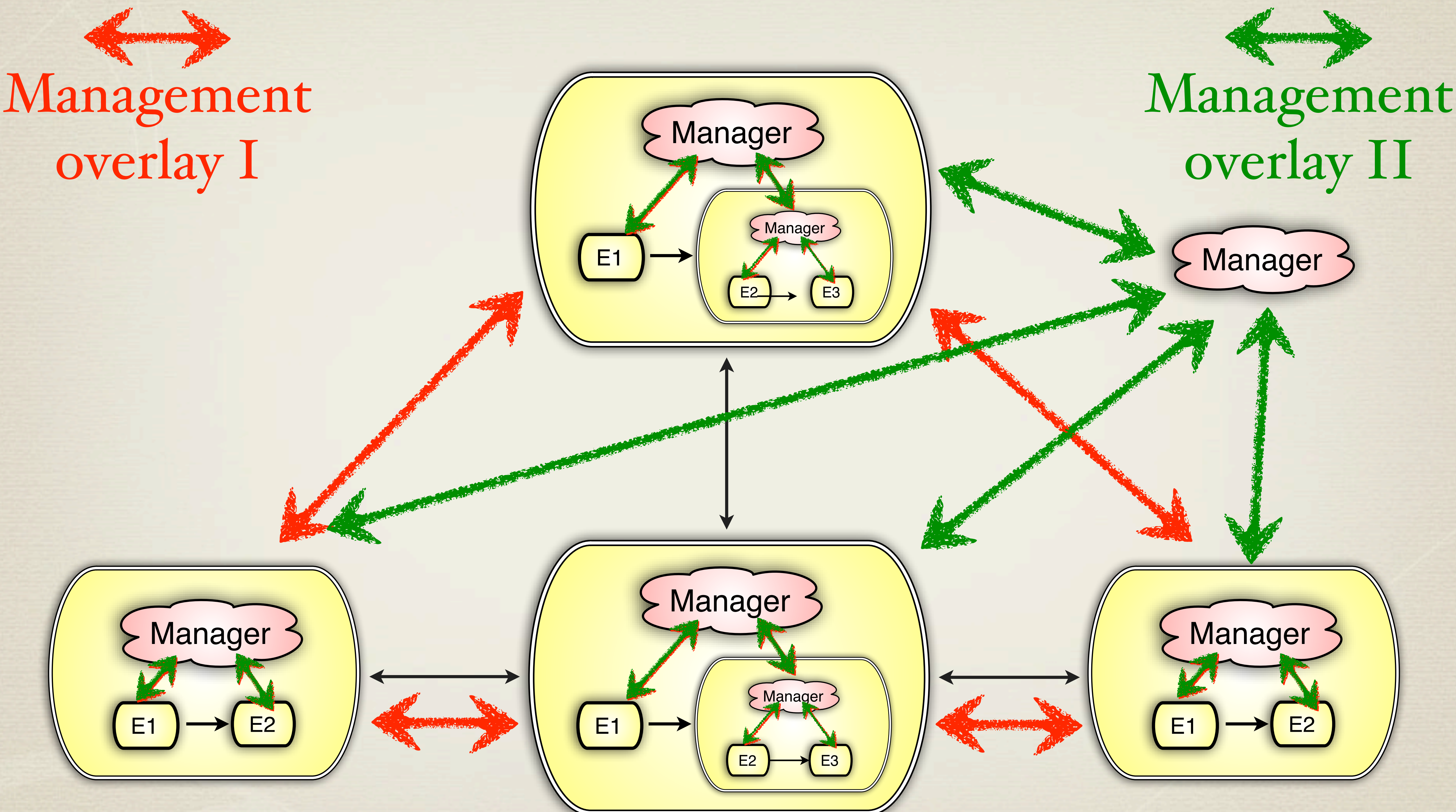
AC - Components in Insulation

* They are components
  * unit of deployment, legacy code, well-defined dependencies, XML-style assembly, etc.
* Autonomic Components exhibit self-* features
  * self-optimising, self-configuring self-protecting, self-healing
* They can have one or more managers
  * we assumed one, since components can be nested the assumption does not break generality

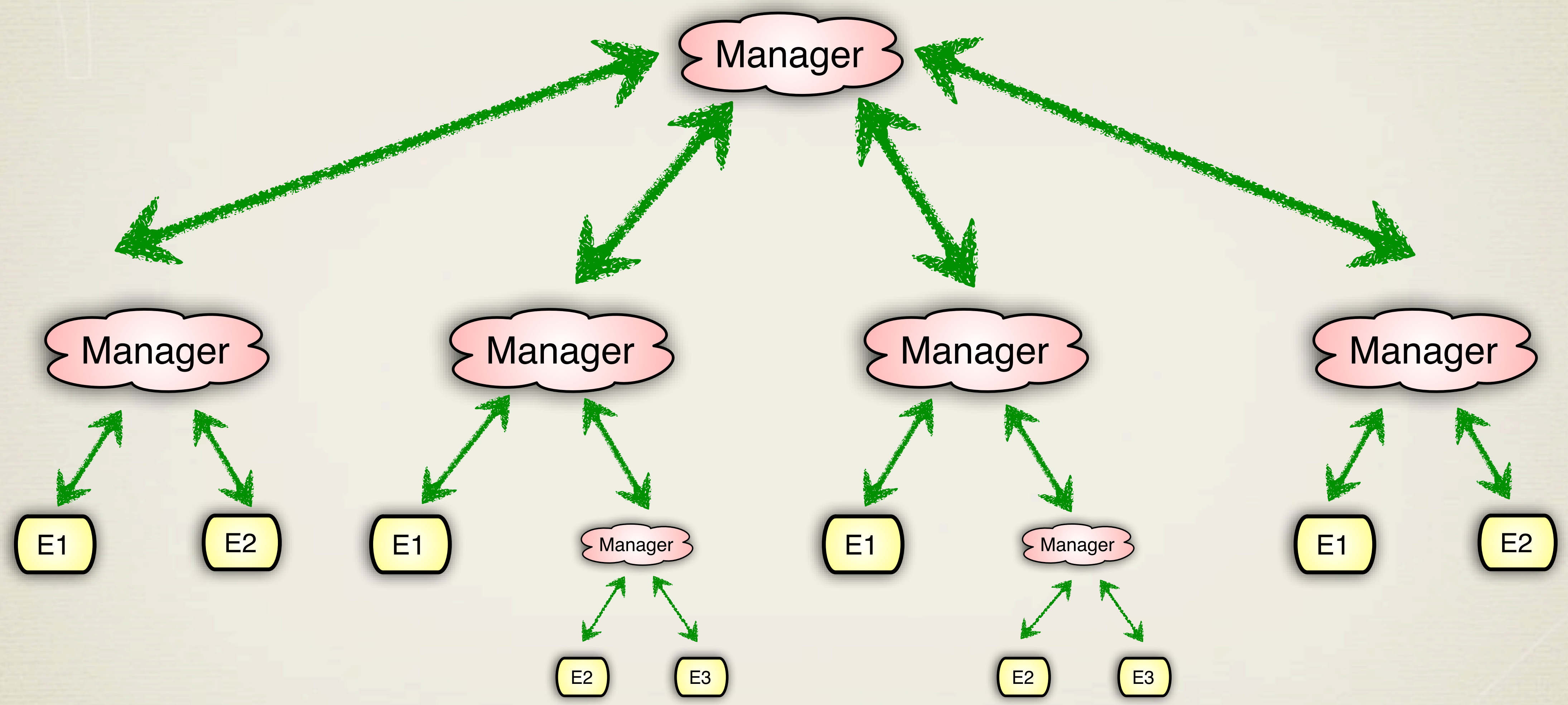
Assembly of Autonomic Components

- Component interaction
  - Legacy (Cl) no interaction, empty manager, no NF ports
  - Passive (Cp) one-way interaction, monitor only capability, read-only NF ports
  - Active (Ca) two-ways interactions, monitor and steering capability, read/write NF ports
- "less general" components can be nested into "more general" components, but not vice-versa
  - $Cl \subset Cp \subset Ca$

Assembly of Autonomic Components

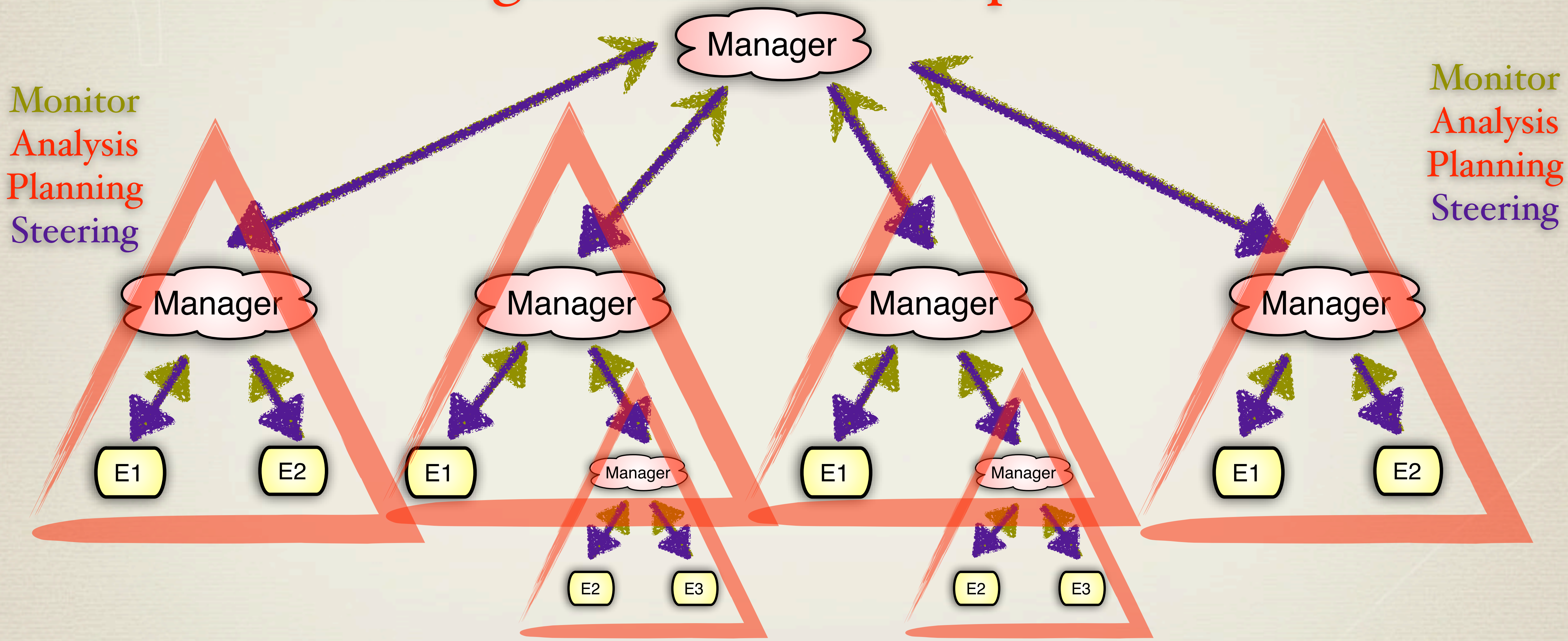


Hierarchy of Autonomic Components



Hierarchy of Autonomic Components

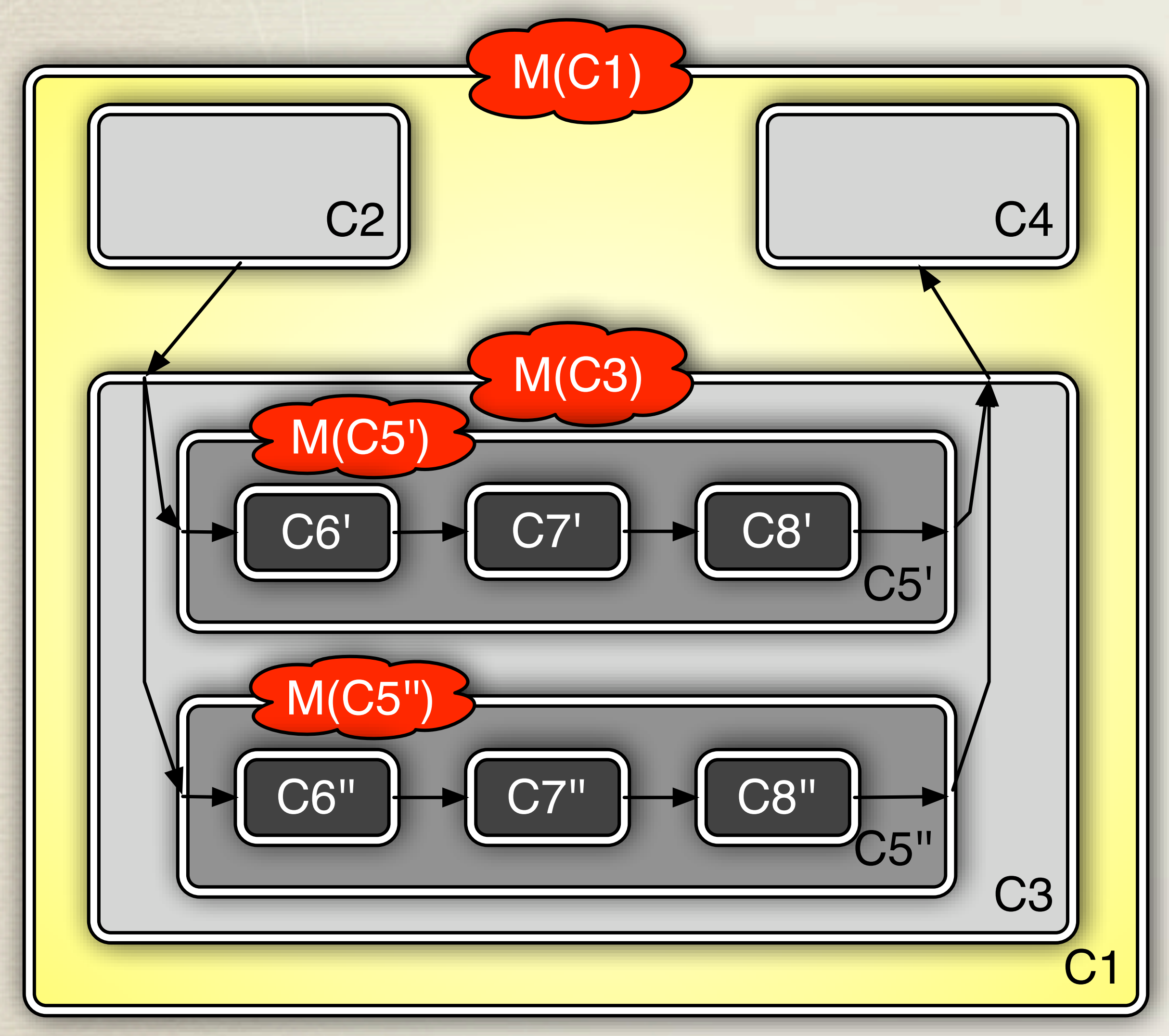
Monitor data is analysed
Management actions are planned



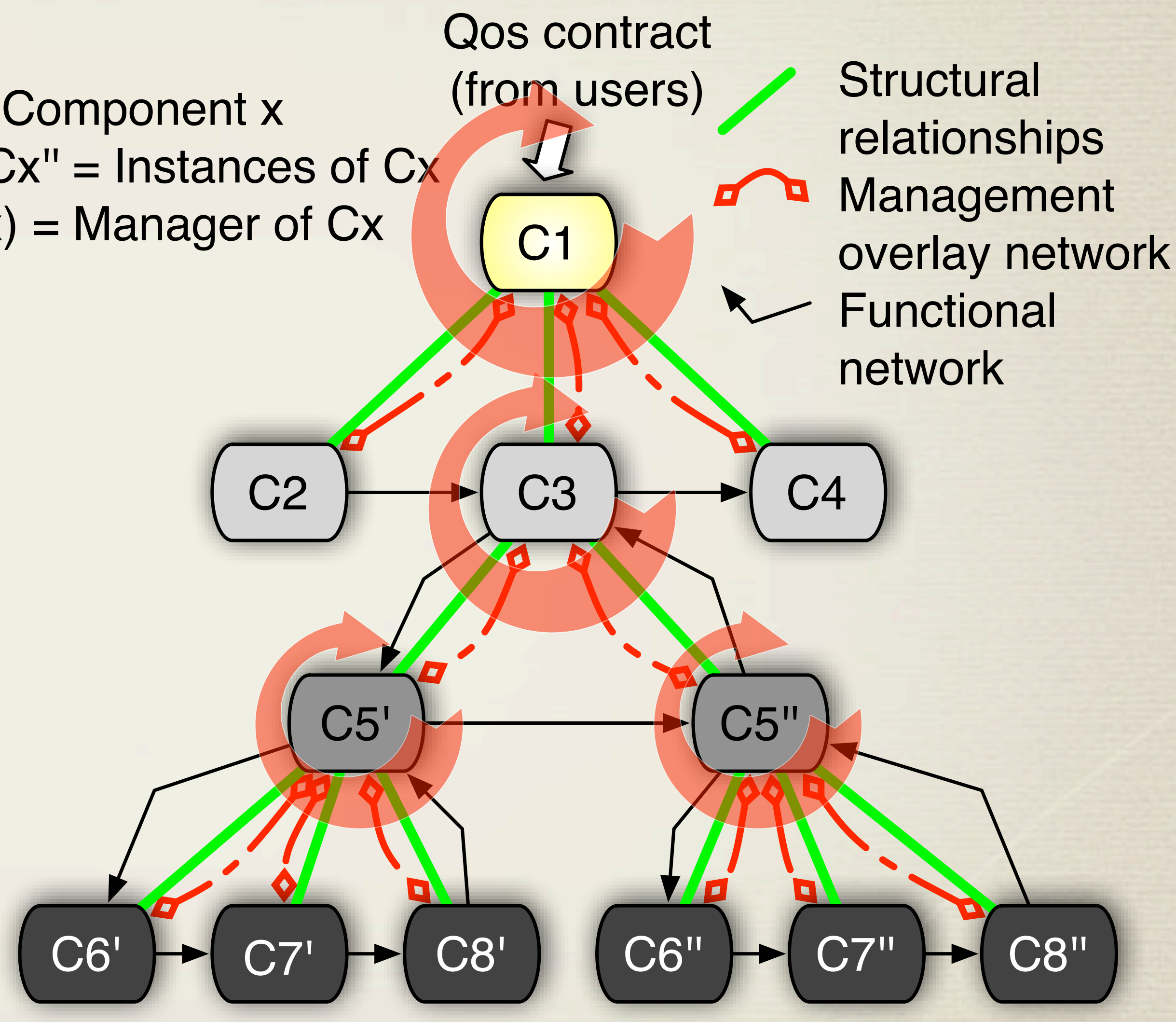
Monitor data (polls/events) can be synthesised (bottom-up)

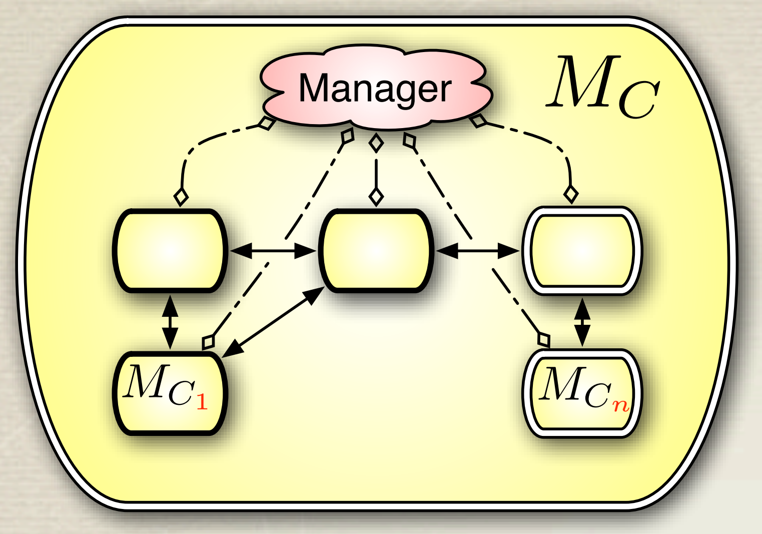
Steering actions and new contracts can be diffused (top-down)

Overlay of managers



Cx = Component x
Cx', Cx'' = Instances of Cx
M(Cx) = Manager of Cx





Monitor + Analyse

Broken contract
goto planning phase

$$e.g.\ (k_{low} \leq {m_1}^1 \leq k_{high}) \wedge {m_2}^1 \leq \theta$$

$$\mathcal{C}ontract\mathcal{P}redicate(\overline{m}_1, \ldots, \overline{m}_n)$$

$$M_C$$

Valid

t = t + 1

$$M_{C_1} \qquad \ldots \qquad M_{C_n}$$

$$\ni \qquad \qquad \ni$$

$$\{{m_1}^1, \ldots, {m_1}^{j_1}\}_{\circlearrowleft_t} \qquad \{{m_n}^1, \ldots, {m_n}^{j_n}\}_{\circlearrowleft_t}$$

$$\overline{m}_1(\circlearrowleft_{\mathbf{t}}) \qquad \overline{m}_n(\circlearrowleft_{\mathbf{t}})$$

Wednesday, February 18, 2009

Plan + Execute (reconfigure) 1/2

- The manager choses a plan among defined ones
  - including the empty plan, i.e. better to do nothing
- A **plan** is composed of
  1. A reconfiguration **protocols** (composed of **actions**)
     - **migrate** C1 on Platform2; **clone** C2 and **wire** it to C1;
     - **actions can also consist in communications with other managers**
  2. **Expected benefit and overhead**
     - quantified as alteration of monitor variables at some future iteration
     - e.g. increase throughput using more resources $m_{\mathbf{i}}(\circlearrowright_{\mathbf{t+k}})$
     - $m_0(\circlearrowright_{\mathbf{t+3}}) = g(m_{\mathbf{1}}(\circlearrowright_t), m_{\mathbf{2}}(\circlearrowright_{\mathbf{t}})), m_{\mathbf{1}}(\circlearrowright_{\mathbf{t+3}}) = f(m_{\mathbf{2}}(\circlearrowright_{\mathbf{t}}))$

Plan + Execute (reconfigure) 2/2

- Which is the better plan?
  - The one that gives the best expected benefit - cost
    - according to a give logic
      - we used first-order logic, but other are viable (e.g. fuzzy)
    - possibly after projecting the n-space of results onto a user-defined goal function
- Is there any guarantee that everything will work as expected
  - No. It is a speculation, but
    - It is control loop theory from the mid of last century
    - We can reach a good sub-optimum by iterating the process
      - this reduces the forecast window
      - this take in account changing enviroment

ON WHY PROGRAMMING AC IS NEARLY A NIGHTMARE
... AND WHY WE INTRODUCED BEHAVIOURAL SKELETONS

Management is Hard to Express 1/3

* AC idea is basically a vision
  * the definition "per se" does not helps so much in designing self-management applications
  * writing a manager is pretty complex
    * should be decoupled and independent from functional code
    * should preserve semantics of functional code
    * should provide effective management capabilities
  * when applied to components it may specialise them too much
    * loosing reusability, that is one of key advantages of components

Management is Hard to Express 2/3

- Expressing managers might be complex
  - User goals are often multi-purpose
    - Performance: the app should sustain x transactions per second; the app should complete each transaction in t seconds
    - Security: the link between P_1 and P_2 should be secured with k-strong encryption; the DB service is exposed by platform P_3
    - Fault-tolerance: the parallel server should survive to the failure of y platforms
  - User wishes are referred to a dynamic world
    - ... consider that $x, t, P_1, P_2, P_3, k, y$ can dynamically change as may dynamically change the performance and the state of the running environment

Management is Hard to Express 3/3

* Ideal application management is distributed
  * but user wishes (goal/contracts) are atomically expressed
  * user would not specify how each part of the (evolving) system contribute to their wishes, and how parts compose w.r.t. goal
* The framework previously presented attacks the problem
  * gives you a well-defined methodology
    * monitor can be collected bottom-up, steering proceed top-down, management happens finding the minimum common ancestor in a tree of autonomic components
  * but does not solve the problems in all cases
    * what happens if the application is composed of two parts user wishes cannot be automatically split in two parts?

Behavioural Skeletons (BeSke)

- Represent an evolution of the algorithmic skeleton concept for component management
  - abstract parametric paradigms of component assembly
  - specialised to solve one or more management goals
    - self-configuration/optimization/healing/protection.
  - carry a semi-formal/formal description and an implementation
    - they are component factories, actually
- Are higher-order components
- Are not exclusive
  - can be composed with non-skeletal assemblies via standard components connectors
    - overcome a classic limitation of skeletal systems

Managers Interaction is Well-defined

- Can be formally specified, e.g. using Orc (Cook & Misra)

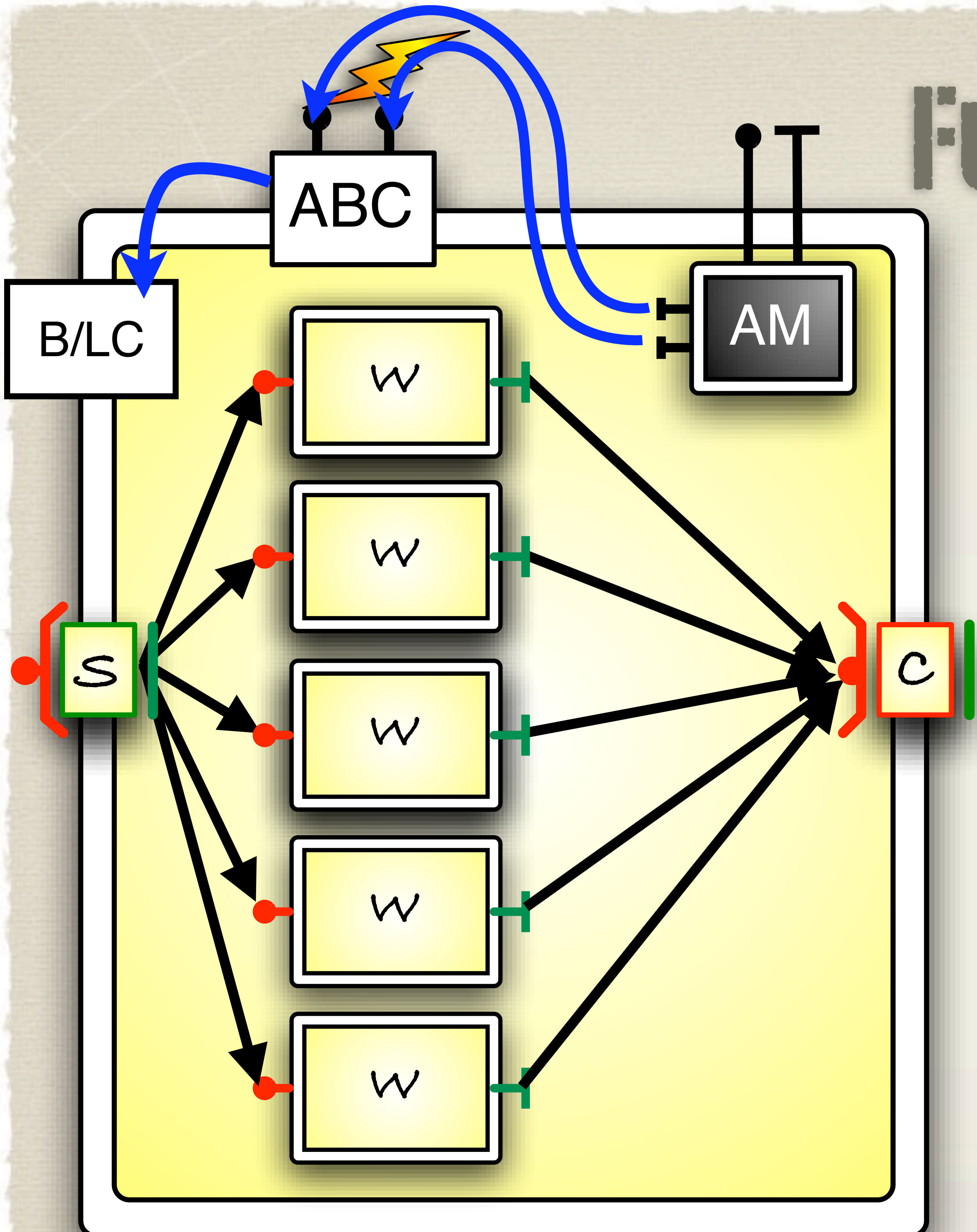
$$BSkel(farm(N), contract) \triangleq$$
$$\quad farm(N) \mid manager(farm(N), contract)$$

$$farm(N) \triangleq \ (\mid 1 \leq i \leq N : W_i)$$

$$W_i \triangleq$$
$$\quad (\ \ \texttt{if}(b) \gg (W_i\_execute(x) > y > out.put(y) \gg W_i)$$
$$\quad \mid \texttt{if}(\neg b) \gg 0)$$
$$\qquad \texttt{where} \ \ (x, b) :\in$$
$$\qquad\quad (\quad in.get > y > let(y, true)$$
$$\qquad\quad \mid Interrupt_i.get > y > let(y, false))$$

$$adapt(farm(N), plan) \triangleq$$
$$\quad (\texttt{if}(plan = addworker) \gg let(y) \gg farm(N+1)$$
$$\qquad \texttt{where} \ \ (\forall i :: y_i :\in Interrupt_i.set)$$

Functional Replication



1. Choose a schema
   e.g. functional replication
   ABC API is chosen accordingly
2. Choose an inner component compliant to BeSke constraints
3. Choose behaviour of ports
   e.g. unicast/from_any, scatter/gather
4. Run your application then trigger adaptations
5. Automatise the process with a Manager

ABC = Autonomic Behaviour Controller (implements mechanisms)
AM = Autonomic Manager (implements policies)
B/LC = Binding + Lifecycle Controller

BeSke families

* Functional Replication
  * Farm/parameter sweep (self-optimization)
  * Stateful Data-Parallel (self-configuring map-reduce)
  * Active/Passive Replication (self-healing)
* Proxy
  * Pipeline (coupled self-protecting proxies)
* Wrappers
  * Facade (self-protection)
* Many others can be borrowed from Design Patterns

Why BeSke are an Advance

- We can associate a standard manager to each BeSke
  - contracts can be predefined, implementation can be **automatically** generated (by way of a factory)
- BeSke are compositional
  - when nested we can **automatically** derive the global behaviour of the assembly that is managed in fully distributed way
  - they can be wired in arbitrary graphs
    - in this way the previous property is not always true
- A prototypal implementation exists (GPL)
  - download from my home page http://www.di.unipi.it/~aldinuc
  - managers implemented as JBoss engines, see references

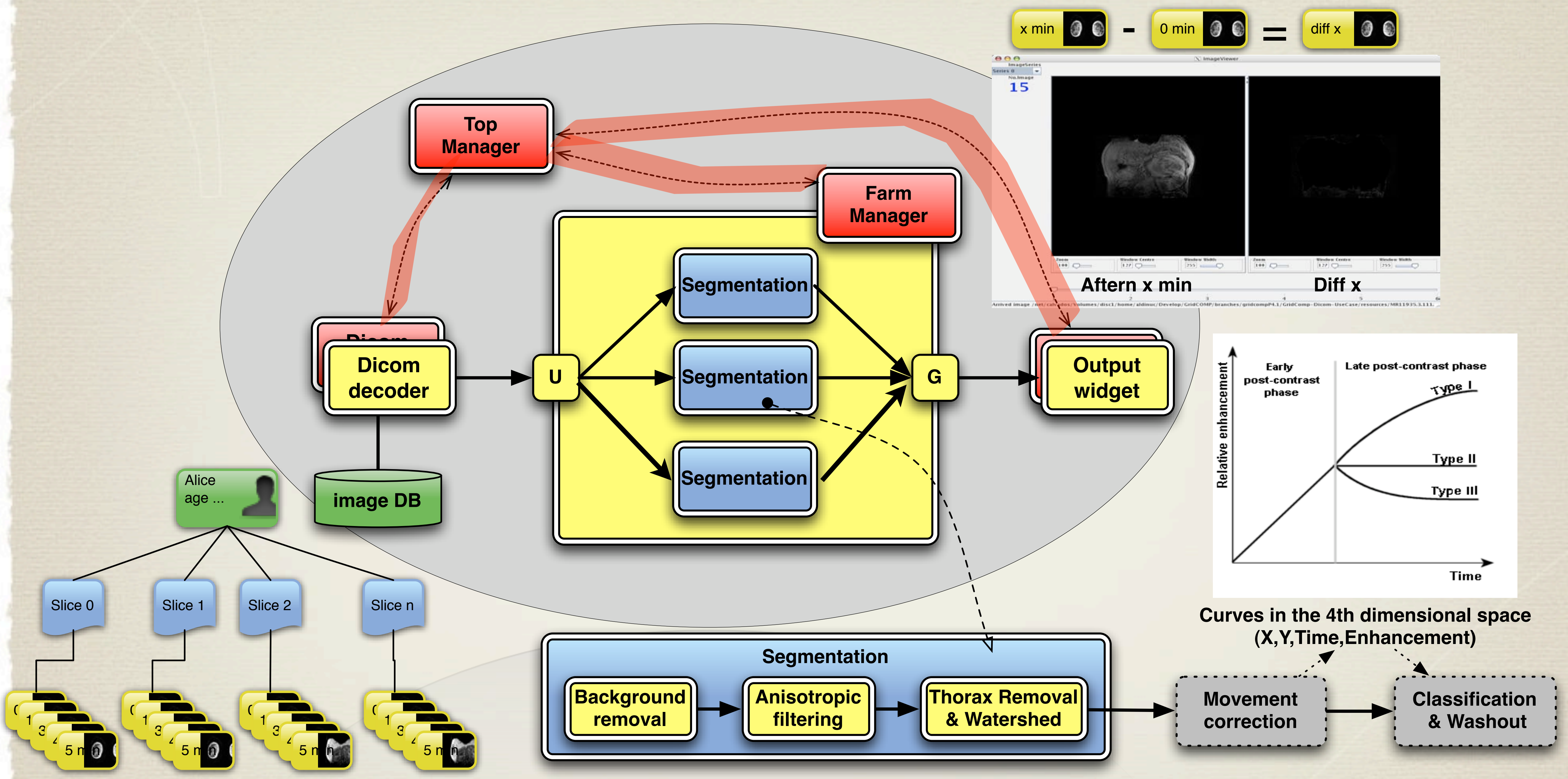
Managers and Contracts

Component	Type	Manager Contract	m_i
C_1	active pipe	$K_{\text{low}} \leq T_{\text{self}} \leq K_{\text{high}}$ (*user defined*)	$K_{\text{low}}, K_{\text{high}}$ constants; $T_{C_2}, T_{C_3}, T_{C_4}$ monitored $T_{\text{self}} = \max\{T_{C_2}, T_{C_3}, T_{C_4}\}$ [↑] $\mathcal{CP}_{C_2} = \mathcal{CP}$; $\mathcal{CP}_{C_3} = \mathcal{CP}$; $\mathcal{CP}_{C_4} = \mathcal{CP}$ [↓]
C_3	active farm	$(\mathcal{CP}_{\text{super}}) \wedge (IT_{\text{self}} \leq T_{\text{self}})$ (*derived*)	IT_{self} = request inter-arrival time; n_{self} = #workers let C_j children of $C_3, 1 \leq j \leq n_{\text{self}}$: T_{C_j} monitored $T_{\text{self}} = \sum_{j=1..n_{\text{self}}} T_{C_j}/n_{\text{self}}^2$; [↑] $\mathcal{CP}_{C_j}$ = optimise(T_{C_j}); [↓]
C_5	active pipe	$\mathcal{CP}_{\text{super}}$ (*derived*)	$T_{C_6}, T_{C_7}, T_{C_8}$ monitored $T_{\text{self}} = \max\{T_{C_6}, T_{C_7}, T_{C_8}\}$; [↑] $\mathcal{CP}_6$ = null; $\mathcal{CP}_7$ = null; $\mathcal{CP}_8$ = null; [↓]
$C_{2,4,6,7,8}$	passive seq	none	provide $T_{C_{2,4,6,7,8}}$ via NF port (respectively)

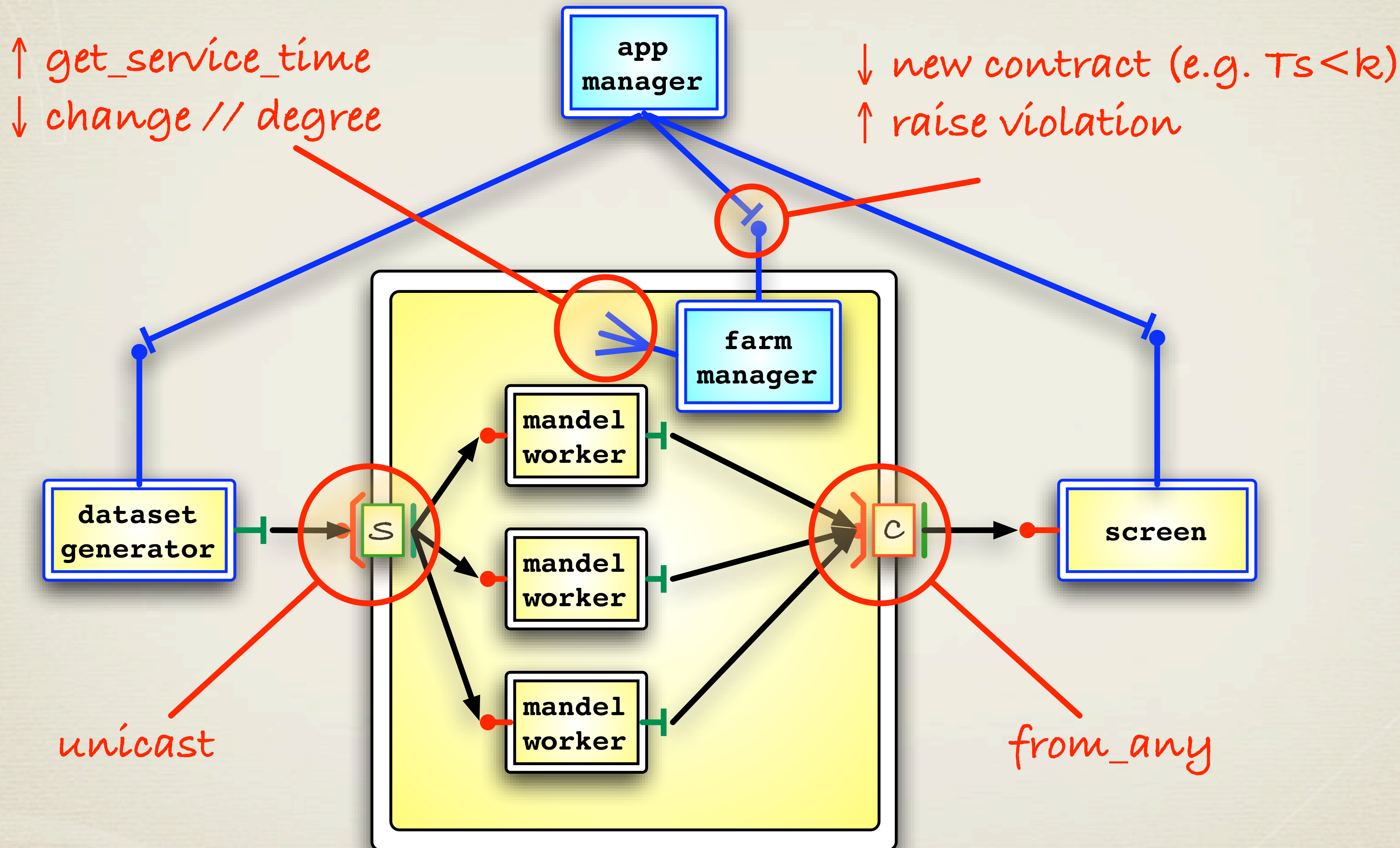
	Plan	Expected Cost	Expected Benefit
PL_{F_1}	move the slower worker C_w to a faster platform, if any	cost(stop(C_w); deploy(C_w); start(C_w))	decrease service time. $T_{\text{farm}}(\circlearrowleft_{t+h}) = \delta T_{C_w}(\circlearrowleft_t)$, $0 \le \delta \le 1$ speed difference between the platforms
PL_{F_2}	increase parallelism degree (allocate k new workers)	cost(deploy(C_{w_j}); start(C_{w_j})) for $j = 1..k$ instances	decrease service time. $T_{\text{farm}}(\circlearrowleft_{t+h}) = \delta T_{\text{farm}}(\circlearrowleft_t)$ $\delta = n/(n+k)$
PL_{F_3}	decrease parallelism degree (de-allocate k workers)	cost(stop(C_{w_j})) for $j = 1..k$ instances	increase service time. $T_{\text{farm}}(\circlearrowleft_{t+h}) = \delta T_{\text{farm}}(\circlearrowleft_t)$ $\delta = (n+k)/n$
PL_{F_4}	raise violation	0 *(negligible)*	none
PL_{P_1}	move stage (C_s) with maximum T to a faster resource, if any	cost(stop(C_s); deploy(C_s); start(C_s))	decrease service time. $T_{\text{pipe}}(\circlearrowleft_{t+h}) = \delta T_{\text{pipe}}(\circlearrowleft_t)$, $0 \le \delta \le 1$ speed difference between the platforms if $max\{T_{C_S}, T_{\text{pipe}}(C_1, \ldots, C_{s-1}, C_{s+1}, \ldots, C_k\} = T_{C_S}$, otherwise $\delta = 1$
PL_{P_2}	collapse adjacent stages C_s, C_{s+1}	cost(stop(C_s); deploy(C_s); start(C_s)) for C_s and C_{s+1}	decrease resource usage $n = n-1$. increase service time. $T_{\text{pipe}}(\circlearrowleft_{t+h}) = \delta + T_{\text{pipe}}(\circlearrowleft_t)$, $\delta = 0$ iff $T_{C_s} + T_{C_{s+1}} \le T_{\text{pipe}}(\circlearrowleft_t)$, $\delta = T_{C_s} + T_{C_{s+1}} - T_{\text{pipe}}(\circlearrowleft_t)$ otherwise
PL_{P_3}	raise violation	0 *(negligible)*	none

DEMO

Example: Mammography



Mandelbrot example (two-levels)



demo

Conclusions

- We have outlined a framework suitable for modelling hierarchical autonomic management
  - not only for grid: clouds, distributed , multi/many core, ...
- We enriched the framework with behavioural skeletons
  - previously existing only “in insulation”
  - contracts and manager implementation can be automatically generated also in case of composition
- We implemented (GCM); we got a ETSI standard
  - we show with a demo they are effective
- GridCOMP has been elected in Jan 2009 “EC ITC project of the month”

THANK YOU

Questions?

1. M. Aldinucci, S. Campa, M. Danelutto, P. Dazzi, P. Kilpatrick, D. Laforenza, and N. Tonellotto. Behavioural skeletons for component autonomic management on grids. In CoreGRID Workshop on Grid Programming Model, Heraklion, Crete, Greece, June 2007.
2. M. Aldinucci, S. Campa, M. Danelutto, M. Vanneschi, P. Dazzi, D. Laforenza, N. Tonellotto, and P. Kilpatrick. Behavioural skeletons in GCM: autonomic management of grid components. In D. E. Baz, J. Bourgeois, and F. Spies, editors, Proc. of Intl. Euromicro PDP 2008: pages 54–63, Toulouse, France, Feb. 2008. IEEE.
3. M. Aldinucci and M. Danelutto. Skeleton-based parallel programming: functional and parallel semantic in a single shot. Computer Languages, Systems and Structures, 33(3-4):179–192, Oct. 2007.
4. M. Aldinucci, M. Danelutto, and P. Dazzi. Muskel: an expandable skeleton environment. Scalable Computing: Practice and Experience, 8(4):325–341, Dec. 2007.
5. M. Aldinucci, M. Danelutto, and P. Kilpatrick. Adding metadata to orc to support reasoning about grid programming. In T. Priol and M. Vanneschi, editors, Towards Next Generation Grids (Proc. of the CoreGRID Symposium 2007), CoreGRID, pages 205–214, Rennes, France, Sept. 2007. Springer.
6. M. Aldinucci, M. Danelutto, and P. Kilpatrick. A framework for prototyping and reasoning about grid systems. In Parallel Computing: Architectures, Algorithms and Applications (Proc. of PARCO 2007, Jülich, Germany), volume 38 of NIC, pages 235–242, Germany, Dec. 2007.
7. M. Aldinucci, M. Danelutto, and P. Kilpatrick. Management in distributed systems: a semi-formal approach. In A.-M. Kermarrec, L. Bougé, and T. Priol, editors, Proc. Euro-Par 2007 Par[illegible], France, Aug. 2007. Springer.
8. M. Aldinucci, M. Danelutto, P. Kilpatr[illegible] models to di[illegible]code. In S. Gorlatch, P. Fragopoulou, and T. Priol, editors, Grid Computing: Achiev[illegible] pages 1[illegible]24. [illegible]8.
9. M. Aldinucci, M. Danelutto, P. Kilpatri[illegible] Dazzi. [illegible] models to distributed grid Java code. In S. Gorlatch, P. Fragopoulou, and T. Priol, editors, Proc. of the Integrated Research in Grid Computing Workshop, CoreGRID, pages 2–13, Hersonissos, Crete, Greece, Apr. 2008.
10. M. Aldinucci, S. Campa, M. Danelutto, P. Dazzi, P. Kilpatrick, D. Laforenza, and N. Tonellotto. Behavioural skeletons for component autonomic management on grids. In Making Grids Work, CoreGRID, chapter Component Programming Models, pages 3–16. Springer, Aug. 2008.
11. M. Aldinucci and M. Danelutto. Securing skele[illegible] the Muskel experience. Journal of Systems Architecture, 54(9):868–876, Sept. 2008.
12. M. Aldinucci, M. Danelutto, H. L. Bouziane, and C. Pérez. Towards software component assembly language enhanced with workflows and skeletons. In Proc. of the ACM SIGPLAN Component-Based High Performance Computing, pages 1–11, New York, NY, USA, Oct. 2008. ACM.
13. M. Aldinucci, M. Danelutto, G. Zoppi, and P. Kilpatrick. Advances in autonomic components & services. In T. Priol and M. Vanneschi, editors, From Grids To Service and Pervasive Computing, CoreGRID, pages 3–18, Las Palmas, Spain, Aug. 2008. Springer.
14. M. Aldinucci and E. Tuosto. Towards a formal semantics for autonomic components. In T. Priol and M. Vanneschi, editors, From Grids To Service and Pervasive Computing (Proc. of the CoreGRID Symposium 2008), CoreGRID, pages 31–45, Las Palmas, Spain, Aug. 2008. Springer.
15. M. Aldinucci, M. Danelutto, and P. Kilpatrick. Autonomic management of non-functional concerns in distributed and parallel application programming. In Proc. of Intl. Parallel & Distributed Processing Symposium (IPDPS), Rome, Italy, May 2009. IEEE. To appear.
16. M. Aldinucci, M. Danelutto, and P. Kilpatrick. Co-design of distributed systems using skeletons and autonomic management abstractions. In Workshops of Euro-Par 2008, volume 5415 of LNCS, 2009. To appear.
17. M. Aldinucci, M. Danelutto, and P. Kilpatrick. Towards hierarchical management of autonomic components: a case study. In Proc. of Intl. Euromicro PDP 2009: Parallel Distributed and network-based Processing, Weimar, Germany, Feb. 2009. IEEE. To appear.