

ABSTRACT

We propose a new view of object-oriented systems, that allows the refinement of concurrent systems from the abstract object-oriented description of the underlying algorithms down to the register transfer level within one single framework.

The key contribution of this work is the separation of the representation of behaviour and its usage. We introduce a graph based model for the structural representation of concurrent object-oriented systems, that supports alternative behaviours as well as inheritance, polymorphism, reconfiguration and mobility.

1 OBJECTIVES

- Executable internal representation
- Common framework for hardware and software at various abstraction levels
- Object oriented modelling
- Supporting the whole design process
→ behaviour alternatives
- Reconfiguration and mobility

Modelling *usage of behaviour* explicitly

5 ACTORS VS. OBJECTS

Actors

In component based design, actors are usually concurrent, interconnected components, communicating e.g. through channels.

Objects

are rather passive, not explicitly interconnected components, whose behaviour is accessed through abstract "method calls".

Proposed solution

Separate connectivity and behavioral aspects.

8 SEPARATE BEHAVIOUR & DATA

Instance $\hat{=}$ behaviour

- A set of allocated behaviours of *services* and instances of *members* of a certain type.
- Corresponds to a *valid subgraph* of an *implementation graph*, with resolved alternatives and consistently allocated nodes.
- Instances can be "created" during run-time as well, to support dynamic reconfiguration

Object $\hat{=}$ data

- Value/state in a temporal distribution of a certain *type*
- To access the associated services, objects need to be bound to an *instance* (see below)

Inheritance *and* alternative behaviours

2 ALTERNATIVES VS. INHERITANCE

Behaviour alternatives

- Useful during the design process to evaluate different implementation variants.
- Software vs. hardware implementation.
- Different levels of abstraction.

Inheritance

- Encapsulation & Specialisation.
- Using inheritance to model different variants would lead to different *types* with different members etc. \rightsquigarrow unwanted, in general!

Proposed solution

The states/values of the *members* of a *module* should be sufficient to represent the state of an *object*, regardless of its implementation variant.

4 EXAMPLE - Counter

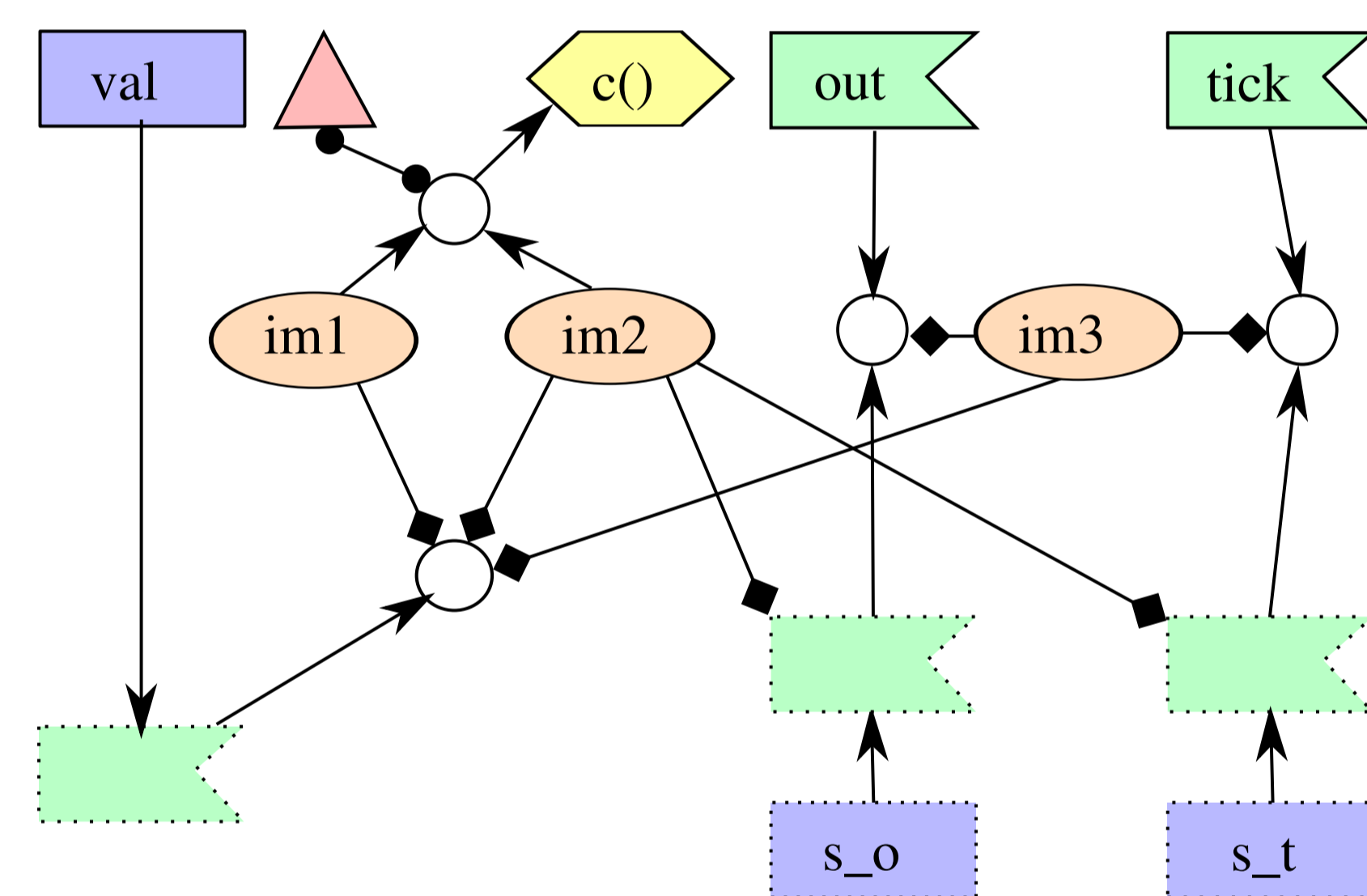


Fig. 2: Example - A counter with alternatives

- The service *c()* increments member *val* every time it is requested.
- Alternatively, the ports *tick* and *out* together with implementation *im3* provide a signal based interface.
- Implementation *im2* of the service uses the signal based interface internally.

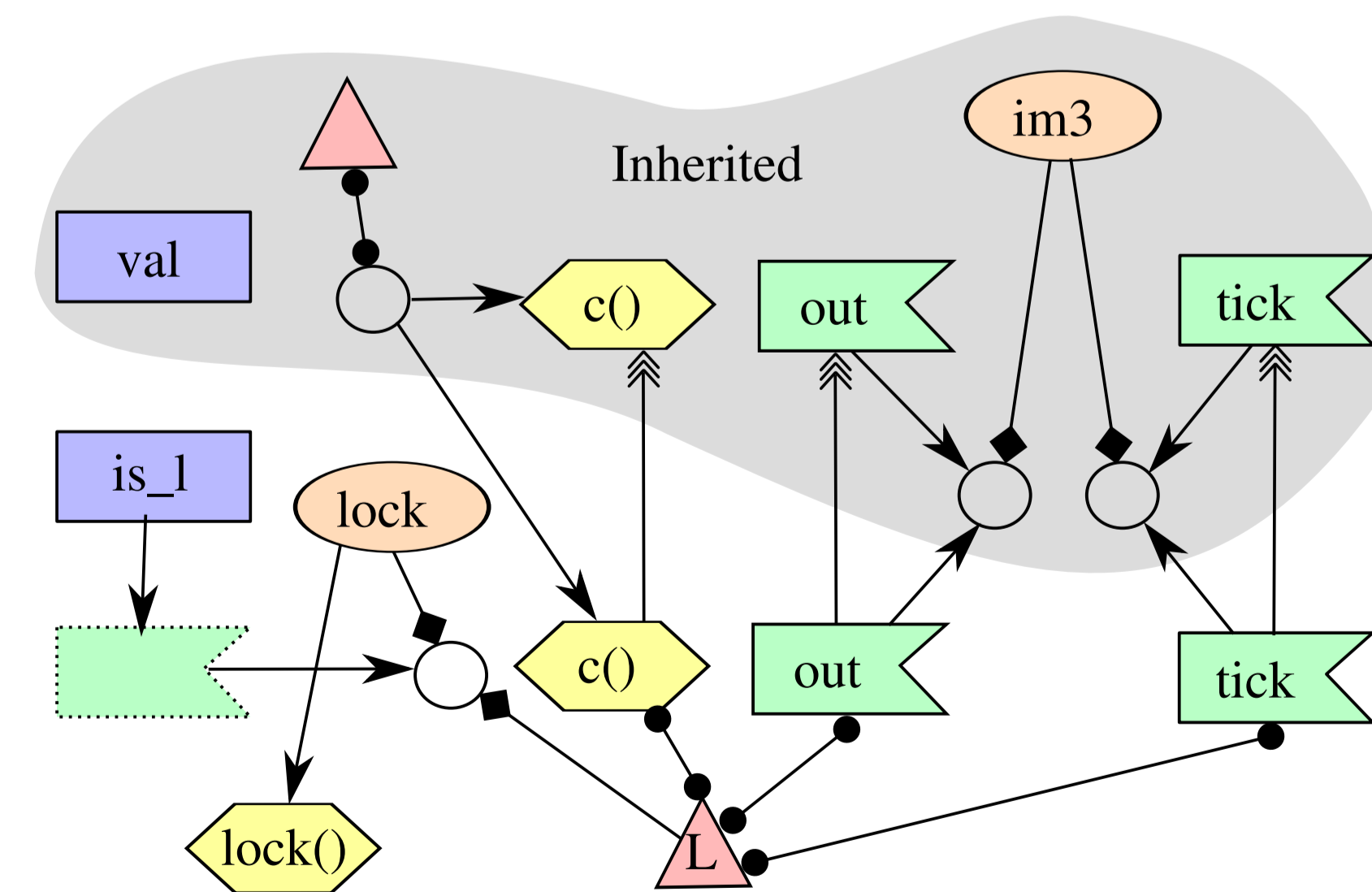


Fig. 3: Example - A lockable counter

- The behaviour is extended with a *lock()* service.
- The existing implementations can be inherited without change.
- Only an additional guard *L* is included, to modify the inherited behaviour.

6 HARDWARE VS. SOFTWARE

Hardware

- Multiple allocation of the same component usually creates corresponding behaviour multiple times (implicitly).
- Resource sharing has to be modelled explicitly and is no integral part of common methodologies.

Software

- Software objects share resources extensively (CPU, common program code in memory, etc.)
- Concurrency leads to conflicting accesses to shared resources. System Level Modelling of additional resources to increase concurrency is difficult.

Proposed solution

Model allocation and usage of behaviour as integral part of the framework → separate data and behaviour.

9 BINDING

Ports → instances

- *Ports* are bound to an *instance*, not to *objects* (see below), to provide separation of connectivity and behaviour.
- Requests to a port are sent to the connected instance, without taking care of the currently bound object
- To support dynamic reconfiguration, port bindings can change during run-time.

Objects → instances

- Objects have to be bound to an *instance* to be accessed
- An *object* can be bound to different *instances* over the time, but not simultaneously (→ mobility)
- Number of *objects*, that can be bound to a certain instance concurrently can be restricted. Simultaneous requests are synchronised through *guards*.
- If multiple objects are bound to an instance concurrently, each object has an associated "position". Then, requests need to include this "address".

3 IMPLEMENTATION GRAPH

Represents inner structure

- Directed, layered graph
- Behaviour and synchronisation of services and its dependencies
- Alternatives modelled via empty nodes and/or multiple edges

Inheritance

- Inheritance is modelled via layers
- Edges between layers model inheritance / overriding
- Synchronisation & implementation can be inherited independently

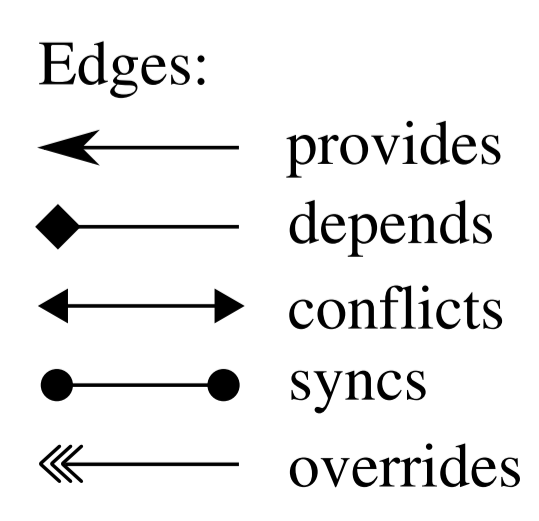
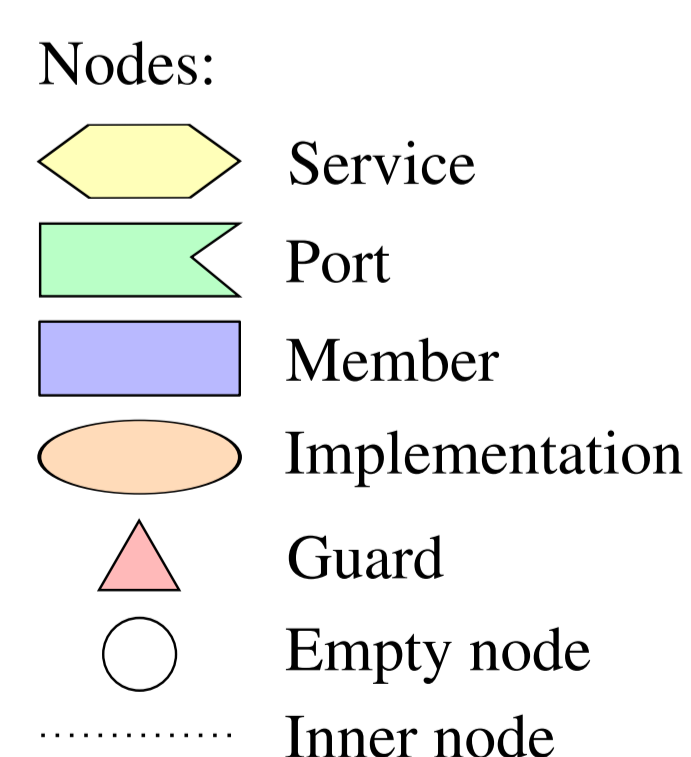


Fig. 1 Node & edge types

7 SERVICE/REQUEST SCHEME

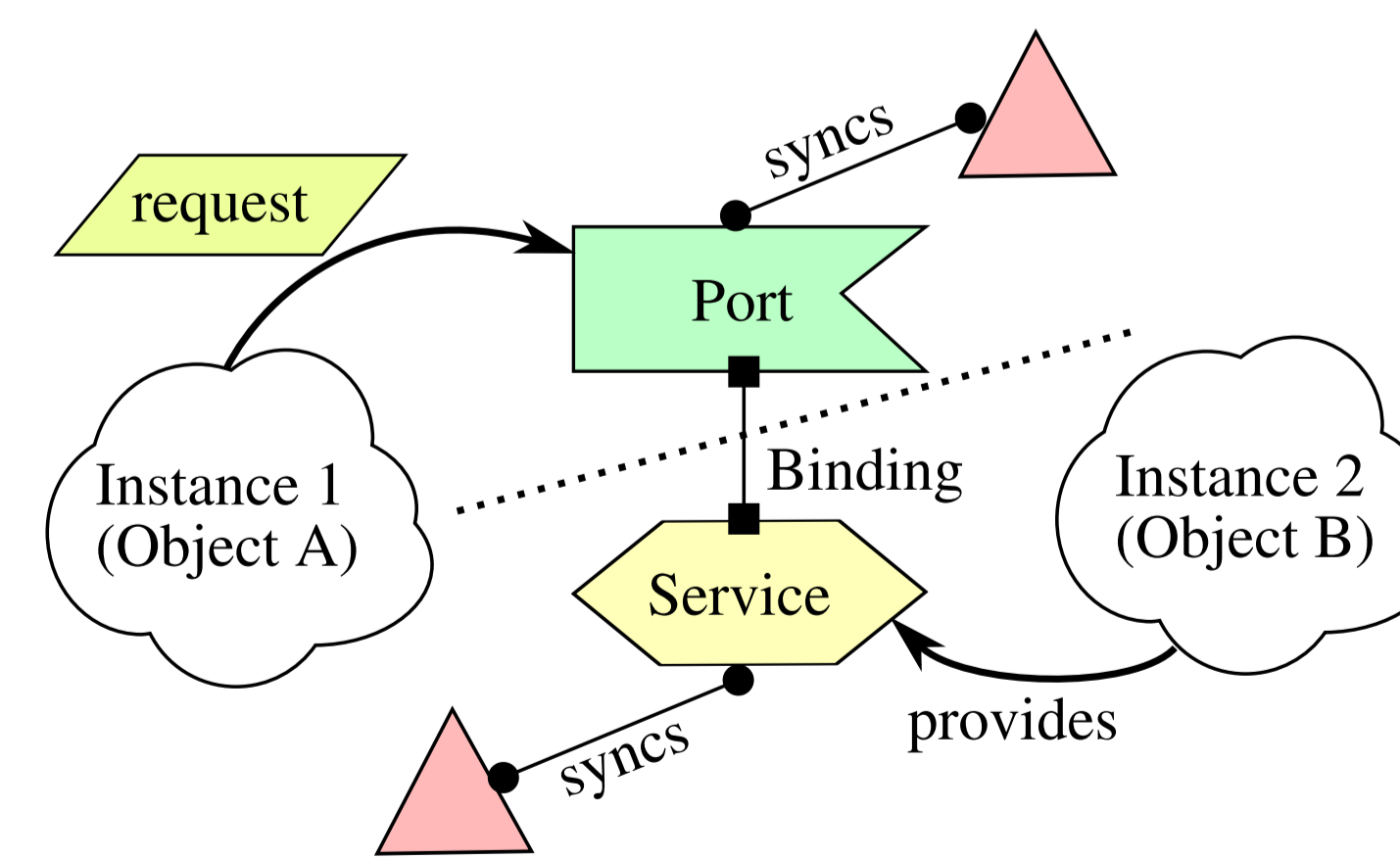


Fig. 4: Simplified service/request scheme

- Behaviour is provided as a set of *services* (may be started concurrently during the *binding*, see below).
- Services are accessed through *requests*.
- *Ports* encapsulate these requests, in order to separate connectivity and behaviour
- Ports and services may have associated *guards* (synchronisation).

10 CONCLUSION

Actors, objects and alternatives

Our approach combines the advantages of object orientation, component based modelling and alternatives within a single framework.

Polymorphism

Polymorphism is supported through *polymorphic instances* (as outlined in the paper).

Reconfiguration & mobility

Everything can be changed during run-time: Create/modify instances and port bindings, create *objects* and bind them to different *instances* over the time.

Pointers and references

If the instance is known, the "positions" of objects bound to the instance are sufficient for access.