Design issues in concurrent object-oriented languages and observability

Thi Mai Thuong Tran and Martin Steffen
Department of Informatics
University of Oslo
Oslo, Norway
{*tmtran,msteffen*}*@ifi.uio.no*

***Abstract*—This paper discusses different choices in the design of object-oriented, concurrent language from the perspective of observability. Observability takes the standpoint that two "program fragments" are observably equivalent if one can be replaced by the other without leading to differences in a larger context. Characterizing the observable behavior of a program fragment is therefore crucial for compositionality.**

The choice of language constructs has a big impact on what can be observed, and thus also how well-suited the language is for being composed. In this paper, we concentrate on well-established variants of constructs in object-oriented languages and discuss their influence on the observable semantics. In particular, we discuss classes as units of code, inheritance as the mainstream way of code re-use in class-based, object-oriented languages. For concurrency, we compare the two most prominent ways to combine objects and concurrency: multi-threading as for instance done in Java vs. the active objects or actor paradigm. A final aspect is the influence of the synchronization mechanism of locks and monitors.

I. Introduction

Compositionality is important in large and distributed systems as it allows programmers to build a larger system from smaller components. In general, a large system cannot be built from scratch by a single programmer or by a company. It often requires a cooperation from different organizations and from different places. In that setting, we need a good mechanism to compose components or replace a component with another one. In general, to replace a component with another, one needs to observe their behaviors to see whether two components are observably equal, if no context can see a difference. One of the most widely adopted solutions is to consider components as black-boxes, and only observe the interactions via their interfaces. So the question is that what can be observed or seen from the outside, from the "client code".

In an object-oriented setting, an open program interacts with its environment via method calls or message exchange. Besides message passing, of course, different communication and synchronization mechanisms exists (shared variable concurrency, multi-cast, black-board communication, publish-and-subscribe and many more). We concentrate here, however, on basic message passing using method calls. In that setting, the interface behavior of an open program C can be characterized by the set of all those message sequences (traces) t, for which there *exists* an environment E such that C and E exchange the messages recorded in t. Thereby we abstract away from any concrete environment, but consider only environments that are compliant to the language restrictions (syntax, type system, etc.). Consequently, interactions are not arbitrary traces $C \stackrel{t}{\Longrightarrow}$; instead we consider behaviors

$$C \parallel E \stackrel{t}{\underset{\bar{t}}{\Longrightarrow}} \acute{C} \parallel \acute{E} \tag{1}$$

where E is a *realizable* environment and trace $\bar{t}$ is complementary to t, i.e., each input is replaced by a matching output and vice versa. The notation $C \parallel E$ indicates that the component C runs in parallel with its environment or observer E. To account for the abstract environment ("there exists an E s.t. ..."), the open semantics is given in an *assumption-commitment* way:

$$\Delta \vdash C : \Theta \stackrel{t}{\Longrightarrow} \acute{\Delta} \vdash \acute{C} : \acute{\Theta} \ , \tag{2}$$

where Δ (as an abstract version of E) contains the *assumptions* about the environment, and dually Θ the *commitments* of the component. Abstracting away also from C gives a language characterization by the set of all possible traces between any component and any environment.

Such a behavioral interface description is relevant and useful for the following reasons. 1) The set of possible traces given this way is more restricted (and realistic) than the one obtained when ignoring the environments. When reasoning about the trace-based behavior of a component, e.g., in compositional verification, with a more precise characterization one can carry out stronger arguments. 2) When using the trace description for *black-box testing*, one can describe test cases in terms of the interface traces and then synthesize appropriate test drivers from it. Clearly, it makes no sense to specify impossible interface behavior, as in this case one cannot generate a corresponding tester. 3) A representation-independent behavior of open programs paves the way for a compositional semantics, a two-level semantics for the nested composition of program components. It allows furthermore optimization of components: only if two components show the same external, observable behavior, one can replace one for the other without changing the interaction with any environment. 4) The formulation gives insight into the semantic nature of the language, here, to different design choices concerning concurrency and object orientation. Some technical material underlying this paper can be found in [1]; here we concentrate on discussing the influence of various design choices from a more practical and global view.

In Section II, we discuss closer observability and the problem of characterizing the interface behavior. Afterwards in Section III resp. IV, we discuss the influence of classes and inheritance, resp. of the concurrency model. In Section V we conclude by summarizing lessons learned from the theoretical approach in more practical terms.

II. Observable behavior

In this section, to give an intuitive understanding of the formal framework, we first sketch in Section II-A the object-oriented calculus which concentrates on the object-oriented features we are interested in; later we extend or modify it by inheritance, by two different concurrency models, and considering synchronization. Afterwards, Section II-B describes the steps of an open semantics, based on the ideas mentioned in the introduction.

A. An object-oriented, concurrent core calculus

The abstract syntax is given in Table I (where *runtime* syntax is underlined). The calculus is rather standard and a class-based variant similar to the object calculi of Abadi and Cardelli [2][3]. The syntax supports objects as instances of classes, local variables, and standard control constructs like conditionals; later we will add concurrency. Objects carry a name or reference, likewise classes and later threads, and via destructive field update, the model supports mutable heap and aliasing.

C	::=	$\mathbf{0} \mid C \parallel C \mid \underline{\nu(n{:}T).C} \mid c[(O)] \mid \underline{o[c,O]} \mid \underline{\natural\langle e\rangle}$	component
O	::=	M,F	object
M	::=	$m = \varsigma(n{:}T).\lambda(\vec{x}{:}\vec{T}).e,\ldots$	method suite
F	::=	$f = fd,\ldots$	fields
fd	::=	$v \mid \bot_c$	field
e	::=	$v \mid \mathsf{stop} \mid \mathsf{let}\, x{:}T = e \,\mathsf{in}\, e$	expressions
	|	$\mathsf{if}\ b\ \mathsf{then}\ e \,\mathsf{else}\, e$	
	|	$v.m(\vec{v}) \mid v.f \mid v.f := v \mid \mathsf{new}\ c$	
v	::=	$x \mid n \mid ()$	values
n	::=	$o \mid c$	names

Table I
Syntax of an OO core calculus

A *component* C is a collection of classes $c[(M,F)]$, objects $o[c,M,F]$, and (for now) one single thread $\natural\langle e\rangle$, with empty component $\mathbf{0}$. The ν-binder is used for hiding and dynamic scoping, as known from the π-calculus [4]. An object o references the class c it instantiates, contains *embedded* the methods it supports plus the fields. The thread $\natural\langle e\rangle$ contains the running code, basically as incarnation of method bodies "in execution". The expression e is basically a sequence of expressions, where the let-construct is used for sequencing and for local declarations. Sequential composition $e_1;e_2$ abbreviates $\mathsf{let}\, x{:}T = e_1 \,\mathsf{in}\, e_2$, where x does not occur free in e_2. The (closed) semantics can be given operationally in a standard way, describing steps of the form $C \xrightarrow{\tau} C'$, modulo standard algebraic laws for parallel composition (such as associativity and commutativity). Being closed, the steps of the semantics are labelled with an internal τ-label, only.

B. Characterizing the observable behavior

Whereas the closed semantics uses internal steps, the open semantics interacts with the environment via communication labels a, and the behavior can be characterized by sequences or traces t of such interaction labels. In an object-oriented setting, the message labels are categorized in method calls, returning the results, and ν-labels which communicate fresh names and which correspond to instantiate a new object from a class. In a concurrent setting later, also fresh thread names are created and communicated via ν-labels. The communication labels are either *incoming* (from the environment to the component) or dually *outgoing* (marked ? resp. !).

As said, without concrete environment, the *open* semantics uses *assumptions* as existential abstraction of all potential environments, and the interaction steps are of the following form:

$$\Delta \vdash C : \Theta \xrightarrow{a} \acute{\Delta} \vdash \acute{C} : \acute{\Theta} \ , \tag{3}$$

(cf. also the traces as sketched in equation (2)). In the step, Δ is the mentioned assumption context, and Θ a commitment context, describing dually relevant interface information about the component C. In software engineering, the terms *provided* and *required* interface are also used instead of commitments and assumptions. In the steps of the open semantics, the two contexts are used as follows: For *incoming* communication, originating from the environment, (mainly) the assumption context Δ is used to *check* whether there *exists* an environment that can send the incoming step (written $\Delta,\Theta \vdash a$ below). The information exchanged over the interface via the communication label a *updates* the (assumption and commitment) contexts in the step (written $\acute{\Delta},\acute{\Theta} = \Delta,\Theta + a$ below).

The steps of equation (3) existentially abstract away from the environment but keep the component part C of the configuration concrete. Abstracting away in the same way from C as well gives a representation-independent characterization of interaction traces possible for well-formed and well-typed open programs. The corresponding judgment

$$\Delta,\Theta \vdash r \rhd t : \mathit{trace}$$

captures the statement: "under assumptions Δ and given commitments Θ, and with history r, the further trace t is possible". The rule of equation (4) inductively formalizes the basic step of that judgment:

$$\frac{\Delta,\Theta \vdash a \qquad \acute{\Delta},\acute{\Theta} = \Delta,\Theta + a \qquad \acute{\Delta},\acute{\Theta} \vdash r\ a \rhd t : \mathit{trace} \qquad \text{other conditions}}{\Delta,\Theta \vdash r \rhd a\ t : \mathit{trace}} \tag{4}$$

The details of the check $\Delta,\Theta \vdash a$ and the update $\Delta,\Theta + a$ as well as the "other conditions" depend on the design decisions concerning the language constructs. We describe the influence of classes, inheritance, two forms of concurrency and related synchronization mechanisms in the following.

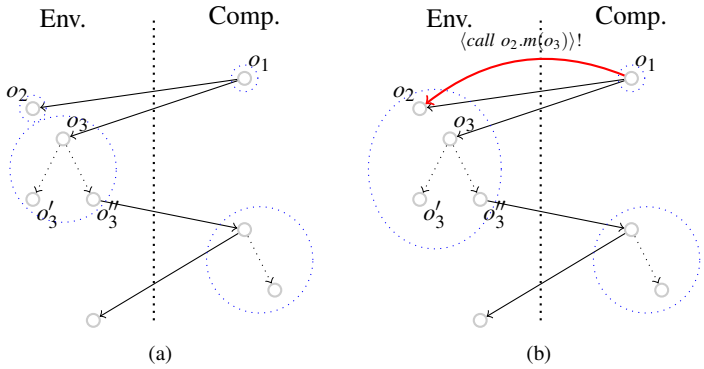

Figure 1. Connectivity

III. Classes and inheritance

Since Simula [5], classes are a central concept in most object-oriented languages. Actually, classes combine different roles in programming languages: 1) they structure the code and offer an abstraction mechanism (as they are also used as type or at least implement an interface). 2) They offer a mechanism of code reuse, typically via inheritance. 3) Finally, they are generators of objects. We describe the influence on the interface behavior of all three roles in turn.

A. Classes as units of composition

In class-based, object-oriented languages, classes describe data together with operations or *methods* to operate on the data and thus provide a programming abstraction, similar to abstract data types [6]. The global heap contains the state of all instances, where objects are identified via their references or addresses, and objects may refer to each other via references stored in their instance variables. When considered as open system, some classes belong to the component, and others are external, i.e., belong to the environment. The state and the methods are therefore only partially part of the component: conceptually, instances of component classes reside in the component part of the heap, and instances of external environment classes are part of the "environment heap", i.e., abstracted away when modelling the behavior of the component.

If in that situation a component object creates an environment object by an instantiation across the border between component and environment, then, without further interface communication, the new environment object cannot be connected to any other object, in the sense that the new object itself cannot contain references to any other object and also that it itself cannot be pointed to by other environment objects. The reason is that all communications which would put the new object in connection in this way would be visible at the interface. For instance, in Figure 1, after the component object o_1 has created the environment objects o_2 and o_3, indicated by the two arrows, it is guaranteed that both o_2 and o_3 are unconnected with each other and no other objects from the environment can point to them. However, o_3 could create environment instances in turn, to which it would be connected, but that would *not* be visible at the interface. The fact that o_3 has created 2 objects *internal* to the environment is indicated by the two corresponding dotted arrows. The (potential) connectivity of objects among each other is important for describing the interface behavior, since certain communications are impossible. For instance, in the situation described so far, no incoming call label a of the form $\langle call\ o.m(o_2,o_3)\rangle?$ (for some callee o) is possible, since no caller object in the environment can point to both o_2 and o_3, as they are necessarily unconnected.

To prevent such impossible interface interaction, the assumption context must contain an over-approximation of such connectivities as an existential abstraction of the heap structure. In particular, the check and update mentioned in the rule of equation (4) must check the connectivity, resp. update that information appropriately.

The potential connectivity of, for instance, environment objects among each other is a reflexive, transitive, and symmetric relation (written $\leftrightharpoons$). We call the equivalence class of potentially connected objects a *clique*. Note that the reflexive, transitive, and symmetric closure over the father-son arrows does *not* apply to the arrows crossing the border. For instance, in the described situation, the check whether o_2 and o_3 are potentially connected, i.e., members of the same environment clique, would fail, i.e.,

$$\Delta,\Theta \nvdash o_2 \leftrightharpoons o_3 \quad \text{but} \quad \Delta,\Theta \vdash o_3 \leftrightharpoons o_3' \leftrightharpoons o_3'' \ . \tag{5}$$

In Figure 1, the arrows show the tree of object creation (with cross-border instantiation as full arrows), and the resulting cliques of objects as dotted bubbles.

The picture so far is static and the cliques are "induced" by the tree of creation. Communication over the interface updates the connectivity information, as formalized by the update $\acute{\Delta},\acute{\Theta} = \Delta,\Theta + a$. For instance, if the component sends an outgoing call $\langle call\ o_2.m(o_3)\rangle!$ with o_1 as caller, o_2 as callee and o_3 as argument, then all four objects o_2, o_3, o_3', and o_3'' are assumed to be connected after the step (see Figure 1(b)).

B. Classes as units of code reuse

Besides describing the implementation of their instances, one common role of classes is that they are units of code reuse via *inheritance*. The most established form

of inheritance is single inheritance, on which we concentrate, even if the results apply to multiple inheritance, as well, only the details get more involved. To represent (single) inheritance, the syntax requires a small addition, only: each class mentions its immediate super-class, i.e., for $c_1[\![(c_2, M, F)]\!]$, c_2 is the super-class of c_1.

In the open semantics in Section III-A, the heap is split into a component heap and to an (abstracted) environment heap, containing instances of component resp. of environment classes. Introducing inheritance means that instances may contain members (fields or methods) whose code is provided by the component as well as ones whose code is provided by the environment. Concerning the *state* of the open system, this existence of component and environment fields in one instance has the following consequence. Not only is the heap separated into component instances on one side and environment instances on the other, now each instance state *itself* is split into two halves, one containing the content of the instance's component fields and the other that of the environment fields.

Figure 2(a) schematically sketches that split when instantiating an instance of a component class c_2 which inherits from an environment super-class c_1. The new object o_2 contains fields from c_2 and from c_1. It is thus depicted as consisting of two halves, where the absent environment half is drawn shaded. Similar to the situation in Section III-A, and without further communication, the *environment* fields of the new object o_2 do not point to any other object and furthermore, o_2 is not pointed at by environment fields of any object. In that sense, the *environment half* of o_2 forms a separate clique, indicated by the dotted circle and unconnected until interface interaction puts it into connection. We assume the standard good practice that fields are "private", so a method added in a sub-class cannot access (via `this`) fields inherited from a super-class. As a consequence of that privacy restriction, component fields can be accessed only by component methods, and analogously for fields and methods of the environment.

A second consequence is that, due to late binding and overriding, seemingly internal implementation details are actually externally observable. One symptom of that is known in software engineering as the fragile base class problem [7]. A base class in an inheritance hierarchy is a (common) super-class, and fragile means that replacing one base class by another, seemingly satisfying the same interface description, may break the code of the client of the base class, i.e., change the behavior of the "environment" of the base class. Consider the following code fragment.

```
class A {                          class B extends A {
  void add () {...}                  void add () {
  void add2 () {...}                   size = size + 1;
  ...                                  super.add(); }
}                                    void add2 () {
                                       size = size + 2;
                                       super.add2();}
```

The two methods *add* and add_2 are intended to add one respectively two elements to some container data structure. Even if informally, this completely describes the intended behavior of A's methods. Class B in addition keeps information about the size of the container. Due to late-binding, this implementation of B is wrong if the add_2-method of the super-class A is implemented via *self*-calls using two times the *add*-method. The problem is that *nothing* in the interface, e.g., in the form of pre- and post-conditions of the methods, helps to avoid the problem. The interface specification is too weak to allow to consider the base class as a black box which can be safely substituted based on its interface specification only. In other words, due to late binding, the version of A where add_2 implements its functionality via seemingly internal self-calls *add* and the version where it directly implements it are observationally different. In the open semantics of the form of equation (3), this is reflected by the fact that the mentioned self-call constitutes a call across the interface; assuming e.g., that A is a component class and the sub-class B an environment class, a self call from method add_2 of A is an outgoing call from the component to the environment if add_2 is executed on an instance of B, as in that case the `this` refers to the environment method *add* of B. Note that the observability of the self-call does not depend on the use of the `super`-keyword (which is used here only to make the example more plausible).

When formulating the open semantics, the assumption and commitment contexts must represent two equivalence relations, one as abstraction of the environment fields and their connectivity and one for the component fields. These abstractions are used to check the interface steps and are correspondingly updated similar to before. Taking the tree of object creation from Figure 1(a), in the setting with inheritance, the clique structure now looks schematically as in Figure 2(b). Since each object is now split into two halves, all objects from the original figure have now a "mirrored" counterpart. For instance, if the component half of o_1 instantiates o_2 and o_3 (as before), the *environment half* of o_1 is connected to both o_2 and o_3 (indicated here by 2 pairs of arrows). Note that the clique structures on the component side and on the environment side differ. For instance, in the figure, the component half of o_3 is connected to o_3' and o_3'', however, for the environment part o_3, o_3', and o_3'' are all members of different cliques. Also in case of communication (as shown in Figure 1(b) in the setting without inheritance), the two clique structures are updated differently. For instance, an outgoing communication merges only cliques at the receiving side of the component, i.e., only environment cliques.

C. Classes as generators of objects

One last aspect of classes we shortly discuss is that classes are *generators* of objects. In particular, two instances of the same class are, until the first differentiating incoming input, identical up to their name or address. That means, that their behavior, when confronted with the same sequence of inputs must be identical (up to the object names). In the description of the open behavior and the possible traces from equation (4), the "other conditions"

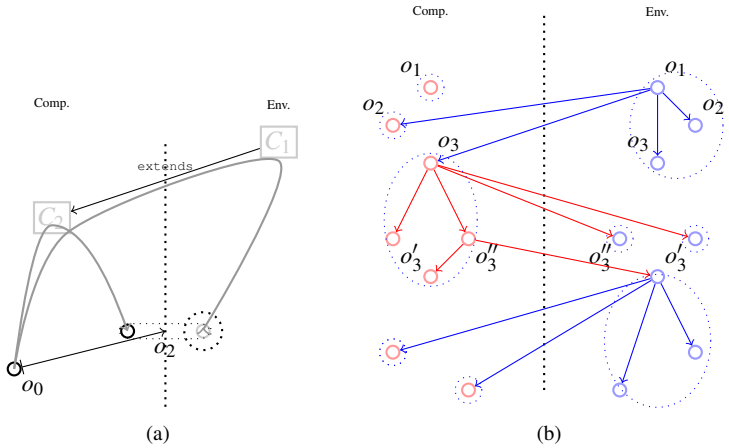

Figure 2. Connectivity and inheritance

must therefore require that the trace is "deterministic" in that sense, i.e., given the history r, the next step a must not contradict behavior seen earlier on an "equivalent" instance of the same class. This restriction of course only applies when the language is deterministic, i.e., in particular in absence of concurrency. Note further that the problem of assuring deterministic reactions from instances rests also on the fact that objects may be *unconnected* from other objects (by being in different cliques). In the presence of *global* variables, for instance, class variables, no newly instantiated object would have "a fresh start" and would therefore not be required to show the same behavior as an earlier instantiation of the same class. Of course the complication does not occur when objects are not created from classes, or each class is instantiated only once. The formal study of full abstraction in a class-less object-oriented calculus [8] basically allows to instantiate objects only once (and besides that disallows cross-border instantiation/inheritance, which therefore avoids these problems.

IV. Concurrency model

How to marry concurrency and object-orientation has been a long-standing issue; see e.g., [9] for an early discussion of different design choices. Actually, the basic distinction already discussed in [9] is between considering threads and objects as separate concepts on the one hand or considering objects conceptually as a unit of concurrency (and state) on the other. The first model is the multi-threading model, prominently represented by object-oriented languages like *Java* [10] and C$^\sharp$, [11]. The alternative is known as *active objects* or actor model. When comparing both models, we assume that part of the model are *locks* as mechanism for concurrency control, i.e., objects are now of the form $o[F,M,L]$ were L is the lock.

A. Multi-threading

The concurrency model, as known from languages like *Java*, separates the unit of state (the objects) from the units of concurrency (the threads).[1] The locks to protect shared data are *re-entrant* locks. To formalize that model means to extend the calculus of Table I by a spawn e-expression plus operations for synchronization such as `wait` and `notify`. In order to characterize the interface behavior, the formalization of legal traces from equation (4) needs to be adapted capturing the following two aspects,

- re-entrant calls, and
- the lock-status.

We discuss the two issues in turn.

1) Re-entrant calls: In the multi-threading model, call-backs and re-entrant calls between component and environment are possible, e.g., an outgoing call by one thread from the component to the environment is not directly answered by the corresponding return giving back the result, but followed by an incoming call. Each thread for a legal trace from equation (4) must therefore be a) strictly alternating wrt. incoming and outgoing communications and b) each return must be preceded by a matching call. This amounts to a *context-free* condition on the interactions of one thread p, where on equation (4), one needs one check of a form $\vdash r \rhd a : wbalanced_p^+$ where p is the name of the thread executing a and "weak balance" formalize the condition just mentioned, requiring that the next interaction a must be incoming for instance ($\vdash r \rhd a : wbalanced_p^-$ is dual).

2) Lock status: In *Java*, each object is equipped with a (re-entrant) lock which can be used to protect the internal state of the object from interference or also for programming synchronized blocks. The actual state of a lock is not observable.[2] However, from the interface trace, under certain circumstances, one can draw the conclusion that a lock of an object is definitely taken. That is the case when a call is answered by a call-back. With the information of a lock being definitely taken, certain

[1] The fact that in *Java* concretely a thread is created as an instance of a specific thread class does not change the conceptual distinction between threads and objects.

[2] In our theoretical calculus we do not include a method as `isLocked()` from Java's `ReentrantLock`-class which allows direct inspection of the lock status.

interface interactions are known then to be impossible and must therefore be excluded form the legal traces. Those conditions complicate the description of the interface behavior considerable [12]. The complications are caused, basically, by the important fact that interface interactions have no *instantaneous* effect on the state. This *decoupling* in the formulation of the open behavior is crucial, because enabledness of a step of, say, the component must *not* depend on the (internal) state of the environment and vice versa. This would contradict a compositional description of the open system. In our setting, e.g., sending a call across the interface is *independent* on the state of the callee's lock, as the lock itself is unobservable.

The issue is illustrated in Figure 3. The scenario of Figure 3(a) shows a trace with interactions of 2 threads (red and blue), where after two incoming calls of the two threads, each one responds with a corresponding return. Since neither the interface interaction in the form of a call takes the lock instantaneously nor a return indicates the exact point of lock release, the behavior of 3(a) is actually possible: since the object lock guarantees mutual exclusion, either p_1 executes its the method body completely before the method body executed by p_2, or conversely. Both *serialized* executions are consistent with the shown interface behavior of 3(a).

The situation of Figure 3(b) is similar, only now the red thread p_1 responds with a call-back instead of a return. Also this scenario possible: The call-back of p_1 makes it observable that p_1 at that point actually holds the lock. The outgoing return of p_2 makes it observable, that at some point in the past, p_2 must have held the lock (but does no longer). A possible serial execution consistent with the shown scenario therefore is that p_2 takes the lock first, releases it again, and afterwards p_1 takes it and holds it till the end of the scenario. Unlike the situation of 3(a), this time the observed behavior imposes an *order* in which the method bodies are entered.

As discussed so far, the constraints on the order derived from the observable interaction depended only on observations concerning *synchronization* via the locks. Object creation and thus the exchange of dynamically created identifiers, e.g., object references, impose another ordering constraint: a value cannot be communicated before it has not been created. Consider Figure 3(c), which shows the same communication pattern as 3(b) except that the first incoming call of thread p_1 sends a freshly created object identifier o as argument, indicated by the binder $\nu(o)$. The shown scenario, where p_2 returns the o in the last interaction is *impossible.* As discussed for 3(b), the serialization constraint concerning the locks enforces that p_2 is executed before p_1. The data dependence concerning o requires that p_1 is executed before p_2.

In summary, to capture the legal behavior in the presence of lock synchronization and re-entrant multi-threading concurrency, the conditions of equation (4) need to keep track of the mentioned order constraints and the rule need to check that the order constraints remain *acyclic.*

B. Active objects

An alternative to the multi-threaded model of concurrency is one based on *active objects,* where the object is not only the unit of (encapsulated) state but also a unit of concurrency. One way to move from multi-threading to active objects is to replace standard method calls $v.m(\vec{v})$ by asynchronous method calls, written say $v@m(\vec{v})$. In an asynchronous call, the caller can proceed concurrently with the called method and get the result back from the call only if and when it needs it. In this way, each asynchronous method calls spawns an new thread; hence there is no stack of method calls. For the interface behavior that means the balance condition of Section IV-A1 is not needed, resp. it degenerates to check that there is no return without prior call and the condition degenerates from a *context-free* restriction to a *regular* one (per thread). Also, in the rule of equation (4), when checking $\Delta, \Theta \vdash r \rhd t : trace$, the history r needs no longer be remembered, when formalizing the check for possible traces.

As in the multi-threaded setting the encapsulated state of an active object needs to be protected against unwanted interference. Without re-entrance, that can be achieved by simple *binary* locks, as opposed to re-entrant locks. Likewise as before, the actual lock status is not directly observable and in particular an interface interaction (still) does not indicate the instantaneous acquisition (in case of a call) or release (in case of returning the value) of a lock. Without re-entrance and call-backs, the scenario shown in Figures 3(b) and 3(c) is not possible, the one of 3(a) of course is. Remember that in Figure 3(b) it is the call-back which makes the fact observable that the lock is actually taken. In 3(a), no information about the *current* lock status can ever be observed, which means the lock status needs not be represented when formalizing the legal traces of an open system which simplifies the description considerably. This corresponds to the situation with active objects.

Both facts —regular behavior is simpler than context-free and non-observability of the internal lock status for active objects— are a clear formal indication that the active object models with its simpler interface behavior are better suited for open systems and when considering compositionality.

V. Conclusion

In this paper we discussed issues for object-oriented, class-based languages from the perspective of compositionality and observable behavior. We gave an overview of how design decisions in an object-oriented language influence the description of the black box behavior when considering classes as units of composition. The question of observable black-box behavior can be and has been studied theoretically. Apart from the theoretical problems, there are also practical lessons to be learned: a clean and simple description of the component behavior is an indication that the chosen constructs are suitable for modular design and compositional reasoning. Or conversely, if the open semantics makes clear that the behavior directly or

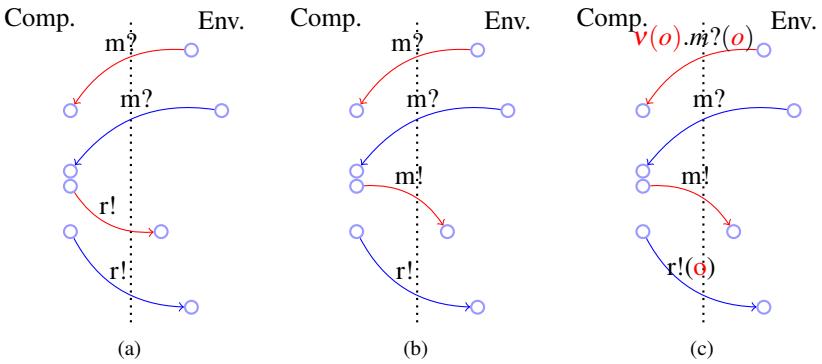

Figure 3. Locks

indirectly exposes internal details at the interface, this is an indication of an inherently non compositional design, for instance not providing enough encapsulation, resp. that the interfaces are too abstract to reflect the reality of what actually is observable.

Based on the sketched theoretical results, we opine the following three main points.

1) Object-orientation and modularity: Using classes as units of composition and as generators of objects exposes an abstract representation of the heap (in the form of connectivity of objects) as part of the interface behavior, which is a considerable complication. Taking into account also inheritance across component boundaries the description becomes even more involved. Basically, classes and sets of objects are no good units of composition, especially if it is allowed to instantiate instances of classes from another component or if inheritance between component borders is possible.

2) Concurrency: The comparison between the two main competing models of concurrency for object-oriented programs in Section IV clearly showed that the multi-threading model is unsuitable as interaction model between components. Components are better considered as communicating asynchronously. Furthermore, the discussion in Section III-C indicates that components should be considered as inherently concurrent from the start since assuming a sequential model actually *complicates* the description of the interface behavior. In other words, a concurrent model of interaction surprisingly simplifies composition.

3) Synchronization: The presence of concurrency requires concurrency control. Especially in connection with multi-threading, the (in principle unobservable) status of the lock may sometimes be inferred by interacting with an object. This fact further complicated the multi-threaded setting by introducing some order constraints. One underlying reason for that seems to be that lock based synchronization is a rather low-level means of guarding against interference. The purpose of using locks is to protect critical regions against unwanted interference, but the way it's achieved is by low-level lock acquisition and release on shared locks. A more compositional, declarative, and high-level way on the user level to achieve protection would be based on *transactional* constructs. Known long from databases, such constructs have recently been proposed as user-level constructs for programming languages, for instance for Java [13] as well as for other languages. We leave the study of observational semantics for such designs as future work.

References

[1] M. Steffen, "Object-connectivity and observability for class-based, object-oriented languages," Habilitation thesis, Technische Faktultät der Christian-Albrechts-Universität zu Kiel, Jul. 2006, 281 pages.

[2] M. Abadi and L. Cardelli, *A Theory of Objects*, ser. Monographs in Computer Science. Springer-Verlag, 1996.

[3] A. D. Gordon and P. D. Hankin, "A concurrent object calculus: Reduction and typing," in *Proceedings of HLCL '98*, ser. Electronic Notes in Theoretical Computer Science, U. Nestmann and B. C. Pierce, Eds., vol. 16.3. Elsevier Science Publishers, 1998.

[4] R. Milner, J. Parrow, and D. Walker, "A calculus of mobile processes, part I/II," *Information and Computation*, vol. 100, pp. 1–77, Sep. 1992.

[5] O.-J. Dahl, B. Myhrhaug, and K. Nygaard, "(simula 67) common base language," Norsk Regnesentral (Norwegian Computing Center), Oslo, Norway, Technical Report S-2, May 1968.

[6] W. Cook, "Object-Oriented Programming Versus Abtract Data Types," in *Foundations of Object-Oriented Languages (REX Workshop)*, ser. Lecture Notes in Computer Science, J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, Eds., vol. 489. Springer-Verlag, 1991, pp. 151–178.

[7] L. Mikhajlov and E. Sekerinski, "A study of the fragile base class problem," in *Proceedings of the 12th European Conference on Object-Oriented Programming (ECOOP'98), Brussels, Belgium*, ser. Lecture Notes in Computer Science, vol. 1445. Springer-Verlag, 1998, pp. 355–354, a longer version has been published as Turku Centre of Computer Science Technical Report TUCS Nr. 117, June 1997 under the title "The Fragile Base Class Problem and Its Solution".

[8] A. Jeffrey and J. Rathke, "A fully abstract may testing semantics for concurrent objects," in *Proceedings of LICS '02*, IEEE. Computer Society Press, Jul. 2002, pp. 101–112.

[9] P. America, "Issues in the design of a parallel object-oriented language," *Formal Aspects of Computing*, vol. 1, no. 4, pp. 366–411, 1989.

[10] J. Gosling, B. Joy, G. L. Steele, and G. Bracha, *The Java Language Specification*, Second ed. Addison-Wesley, 2000.

[11] *C# Language Specification*, 2nd ed., ECMA International Standardizing Information and Communication Systems, Dec. 2002, standard ECMA-334.

[12] E. Ábrahám, A. Grüner, and M. Steffen, "Abstract interface behavior of object-oriented languages with monitors," in *FMOODS '06*, ser. Lecture Notes in Computer Science, R. Gorrieri and H. Wehrheim, Eds., vol. 4037. Springer-Verlag, 2006, pp. 218–232 (15 pages).

[13] S. Jagannathan, J. Vitek, A. Welc, and A. Hosking, "A transactional object calculus," *Science of Computer Programming*, vol. 57, no. 2, pp. 164–186, Aug. 2005.