On the Behavioral Inheritance of State-Based Objects

David Harel*
The Weizmann Institute of Science

Orna Kupferman†
The Hebrew University

March 2000

Abstract

We consider the inheritance of state-based behavior in object-oriented analysis and design, as it arises, for example, in specifying behavior in the UML using statecharts. Our goal is to investigate the notion of behavioral conformity and the resulting substitutability of classes, whereby the inheritance mechanism is to retain original behaviors. This is a central issue of crucial importance to the modeling, design and verification of object-oriented systems. There are many deep and unresolved questions around inheritance, which cannot be addressed without a clear and rigorous picture of what exactly is meant by behavioral conformity, and how computationally complex it is to detect. We first define a basic computational model for object-oriented designs, and then define substitutability and inheritance in the linear and branching paradigms. We relate these to trace containment and Milner's notion of simulation and deduce the complexity of some of the relevant algorithmic problems. The paper thus sets the stage for further research on behavioral inheritance.

1 Introduction

Inheritance is a central issue in the object-oriented paradigm. It has been introduced mainly to enable reuse: we want to be able to spend less effort (and to decrease the chance of errors) when respecifying things that have already been specified for a more abstract or more general class. However, reuse is not the only issue; we might also want to make sure that certain things in the system's behavior indeed do not change as a result of the respecification. It is this notion of behavioral inheritance that we wish to investigate here.

The '`is-a`' subclassing relationship between object classes is used to declare inheritance. Intuitively, given two object classes A and B, we say that B is a subclass of A (or B `is-a` A) if whatever we know about A can be inherited to B. For example, minus `is-a` unary operation. The exact meaning of this, however, is not that clear. In virtually all approaches to inheritance

*Department of Computer Science and Applied Mathematics, The Weizmann institute of Science, Rehovot 76100, Israel. Email: harel@wisdom.weizmann.ac.il

†Institute of Computer Science, The Hebrew University, Jerusalem 91904, Israel. Email: orna@cs.huji.ac.il

in the literature, the `is-a` relationship between classes entails a basic minimal requirement of *interface conformity*. This means that we should be able to replace A with B without causing incompatibility. But the mere ability to 'plug in' a B whenever we could have used an A without causing an immediate runtime problem guarantees nothing about the *behavioral conformity* of A and B. In other words, interface, or structural, conformity means that B can be asked to do anything that A can do, and to look as if it is doing what A does, even though it may be actually doing something quite different. Behavioral conformity, on the other hand, means not only that can B be asked to do anything A can do, but that it will do so in the very same way and with the very same results. This notion is not new. It has its roots in work of Wegner and Zdonik [WZ88], and Liskov and Wing [LW94, LW95], and is often referred to as *the Liskov principle*. See also Abadi and Cardelli [AC95]. Niestraz [Nie95] discussed this notion in the framework of state machine modeling, but on a relatively simple level. Our work can be thought of as providing a detailed proposal as to how one might define the Liskov principle in a state-based object model setting (see Harel and Gery [HG97]), and shows its connections with Milner's notions of simulation and bisimulation.

In object-oriented programming languages, behavioral conformity is not much of an issue: when specifying the subclass B we can decide whether to accept or override the code of an operation (the method). In C^{++}, for example, a class derived from a base class can either adopt the bases class's original behavior or can turn it upside down [Str97]. In contrast, in OOAD, high level object-oriented specification methodologies are used, and behavior is not necessarily specified by writing code in a programming language, but by state-based visual formalisms. The one used in most such approaches is the language of *statecharts* [Har87]. This is also true of the recently standardized UML [Uml, HG97, RJB99], a general-purpose modeling methodology that combines the use-case process of OOSE [Jac92] with two popular approaches to object-oriented modeling — the one of Booch [Boo94] and the OMT [Rum91]. In these methodologies, a statechart is attached to a class, and when an instance object of that class is 'alive' the statechart is in execution, controlling both how the object communicates and collaborates with other objects and also how it carries out much of its own internal behavior.

Thus, a typical operation in OOAD will not be implemented simply by a method consisting of a piece of code that can be adopted or overridden, but will be ingrained within the statechart of the class in question, and will often be subtly distributed therein. This means that in OOAD, behavioral conformity becomes a much more complicated issue. What does it really mean to inherit behavior? How complex is the detection of behavioral conformity? What can be done to enforce it? How can we specify that certain 'behaviors' are to be inherited and others not, and what does the phrase 'behaviors' even mean? We believe that the OOAD world should address a variety of such deep issues related to the behavioral inheritance of state-based specification, and should supply engineers with means to help decide what they really want and how to achieve it.

Current definitions of inheritance in the OOAD world have adopted the relatively easy approach of enforcing structural conformity only. In addition, some of them provide simple

restrictions on what can be changed in a statechart when going from a class to an inheriting subclass (see for example [CD94, HG97, Uml]). Specifically, a subclass B of A will be initially based on the same underlying state-transition topology, but with possible refinements — such as new substates or modified targets of transitions — that may enrich A's behavioral capabilities. However, such refinements can easily be shown to have the potential for radically changing the behavior of the class, which may or may not be what the specifier had in mind. Thus, while relaxing behavioral conformity enlarges the scope of inheritance, it places new responsibilities on the specifier in figuring out the behavioral changes that a substitution might entail. Since object-oriented models are intended to be executable and to yield full running code [HG97], *substitutability* and the algorithmic issues it raises become extremely important.

In this paper we define and study such behavioral inheritance. Our first task is to define a basic, rigorous and analyzable model for object-oriented designs. We want it to be powerful enough to capture the essence of object-oriented analysis and design, and to be set up in the spirit of the UML and the tools that support it [Uml, HG97, Rhap, RJB99]. To keep the model's mathematics clean enough for our purposes, we cannot use the full format that a team of engineers would be using on a large real-world project; for example, we will have to compromise on many of the notions and notations that crowd the full UML standard, and which are not defined sufficiently well for our ultimate quest. We believe that a clean and rigorous definition of object-oriented designs is acutely needed, given the multitude of levels of discourse appearing in the literature — propositions vs. predicates with variables, single vs. multiple thread designs, flat sequential state machines vs. statecharts, and symbolic directed messages vs. parameterized messages, to mention only a few. If we are successful in this definitional goal, such a computational model will enable us to provide precise definitions of behavioral inheritance, to prove theorems about them, to import work from other rigorous computational models appearing in the literature (e.g., finite automata, CSP, CCS, and Petri nets), as well as from other design paradigms (e.g., specification, verification, and synthesis), and to compare various formalisms among themselves (e.g., statecharts and live message sequence charts; see, for example, [DH99, HK99]). Results can then be extended and 'lifted up' to the more detailed and less clean realm of real-world modeling languages, such as richer subsets of the UML.

This task is addressed in Section 2, where we define *object systems* and their reactivity. Our definition is basic and simple — in the terminology of logics of programs this would be called a *propositional level* model — but it can be extended without too much trouble. It is also lengthy and rather detailed, but there is no way around this if one wants to be formal enough to actually prove things. This section required the most work on our part, and we regard it as the paper's main contribution. We believe that it constitutes a clean and rigorous conglomeration of many of the bare essentials of object oriented analysis and design modeling. Essentially, an object system specifies a set of classes and requests, and object behavior is given by simple finite state machines (which can be replaced by full statecharts — an extension that is not crucial for our purposes here). An object of a certain class can send single-thread requests to other objects, referring to them by their class name and a direct or indirect index. There are also control

requests that may ask for the creation or destruction of other objects. The semantics is given by a simple run-to-completion rule. Section 5 discusses briefly some of the restrictions we have imposed and related extensions that are possible.

Then, in Section 3, we define what we believe are the right notions of substitutability and behavioral inheritance. We follow two approaches, as in classical work on the verification and specification of concurrency: *linear* and *branching*. In the linear approach, the requirement is that every execution of the system obtained by substituting A by B is a possible execution of the original system (the one with A). In the branching approach, the requirement is that the tree of possible executions of the system obtained by substituting A by B can be embedded in the tree of executions of the original system. Our definition is parameterized by a *refinement mapping*, which can be used to specify the simultaneous substitution of several classes.

These two sections complete the definitions, making it possible to start investigating the resulting notions of substitutability. In Section 4, we take a first step in this direction, by showing the connection between behavioral inheritance and the classical refinement notions of *trace containment* and *simulation* from the literature on the semantics of concurrency [Mil71]. From this connection we deduce the computational complexity of some relevant algorithmic problems, such as deciding whether one class is substitutable for another.

There has been a related effort at defining substitutability in OOAD, which is described in [Nie95, Sou99]. Like our approach here, the idea is to adopt classical refinement relations from the theory of concurrency, but there are differences. In [Nie95], the reduction to the classical notions is achieved by introducing regular types for active objects. In [Sou99] it is achieved by translating behaviors given by state-based visual formalisms (specifically, UML statecharts) to code in an object-oriented programming language. Moreover, the substitutability discussion in [Nie95, Sou99] is less extensive than ours, and is not as in-depth. For example, it uses a simpler domain description (service types in [Nie95] and a simplified form of transition in [Sou99]) and does not address issues like dynamic creation and destruction of objects, or refinement mappings.

There are many reasons to develop a practical treatment of behavioral inheritance; e.g., to enable automatic detection therof, to find syntactic constraints guaranteeing substitutability, and to specify these constraints in languages such as MSCs or LSCs [DH99]. Such a treatement must involve carefully prepared rigorous definitions of a computational model for object-oriented models and designs and for behavioral conformity. Thus, we believe that our work sets the stage for much further research that is essential on these issues.

2 Object Systems and their Semantics

2.1 Classes and objects

We first define the elements that make up the system itself.

A *setting* is a tuple

$$\mathcal{S} = \langle \Sigma, O_1, \ldots, O_n \rangle,$$

consisting of a finite set Σ of *simple requests* and n *classes* $O_1, \ldots, O_n$ of objects, for some $n \geq 1$. We denote the set $\{O_1, \ldots, O_n\}$ of object classes by $\mathcal{O}$. Using the simple requests and the object classes in $\mathcal{S}$, we can define *systems*. A system that is based on $\mathcal{S}$ consists of *instances* of the classes. These instances, which we refer to as the *components* of the system, interact via the simple requests and the more involved requests we will define later. We can have, for instance, a system with three instances of class O_1, no instance of class O_2, and one instance of class O_3. We refer to the different instances of a class by indices in $\mathbb{N}^+$. For example, in the above system, we can denote the participating components by $O_1[1], O_1[2], O_1[3]$, and $O_3[1]$. In order to refer to a particular instance of a class O, we use elements in

$$R = \mathbb{N}^+ \cup \{min, max\}.$$

Thus, a *reference* $\eta \in R$ can be either direct, namely $\eta \in \mathbb{N}^+$, in which case the referred instance is $O[\eta]$, or indirect, namely $\eta \in \{min, max\}$, in which case the referred instance is $O[x]$, for the minimal (respectively maximal) index x that is relevant for the class O. We will later define exactly what "relevant" means.

The evolution of a system involves creation and destruction of components, initiated by requests. Formally, a *control request* is a triple in $\{destroy, create\} \times \mathcal{O} \times R$. Thus, for example, $(create, O_2, 3)$ is a request to create an instance of class O_2 that would get the index 3. A *request* is either a simple request or a control request. We denote the set of all requests by Ω. Thus

$$\Omega = \Sigma \cup (\{destroy, create\} \times \mathcal{O} \times R).$$

Each setting $\mathcal{S}$ induces a set $\mathcal{G}(\mathcal{S})$ of *guards*. Each guard is of the form σ/τ, where $\sigma \in \Omega$ is a request that triggers the guard and $\tau \in (\mathcal{O} \times R \times \Omega)^*$ is a (possibly empty) sequence of *directed requests*. Thus, we might write

$$\mathcal{G}(\mathcal{S}) = \Omega \ / \ (\mathcal{O} \times R \times \Omega)^*.$$

For example, the guard $\sigma_1/\langle O_1, 2, \sigma_4 \rangle \, ; \, \langle O_3, 1, \sigma_2 \rangle$ (for clarity, we add a ';' between directed requests) is triggered by the request σ_1. In order for the guard to be accomplished, the component $O_1[2]$ should fulfill the request σ_4, and, sequentially, the component $O_3[1]$ should fulfill the request σ_2. We term σ the *trigger* of σ/τ, and term τ the *action* of σ/τ. When τ is empty, we omit the /.

We can now define the classes O_i in more detail. Each class O_i in $\mathcal{S}$ is a finite state machine, given in the form of a tuple

$$O_i = \langle Q_i, Q_i^0, \delta_i, \beta_i \rangle,$$

where:

- Q_i is a finite set of states, and we require that the Q_j be pairwise disjoint.

- $Q_i^0 \subseteq Q_i$ is a set of initial states.
- $\delta_i \subseteq Q_i \times \mathcal{G}(\mathcal{S}) \times Q_i$ is a transition relation, each element of which relates a source state with a target state via a labeling guard.
- $\beta_i \in \mathbb{N}^+ \cup \{\infty\}$ is a bound on the number of occurrences (different instances) of O_i in a system based on the setting $\mathcal{S}$. When $\beta_i \in \mathbb{N}^+$, we say that the class O_i is *bounded*. Otherwise (i.e., when $\beta_i = \infty$), O_i is *unbounded*.

We say that a class O_i is *deterministic* if for all states $q \in Q_i$ and requests $\sigma \in \Omega$, at most one transition from q is triggered by σ.

The graphical representation of a class is by standard state diagrams. In labeling the transitions, we adopt the notational convention of using '$*$' as a wildcard, ranging over all possible values. For example, a transition labeled $\langle q, (create, O_i, *), q' \rangle$ stands for the infinitely many transitions $\langle q, (create, O_i, \eta), q' \rangle$ with $\eta \in R$.

For a state $q \in Q_i$, we denote by $into(q)$ the set of guards σ/τ labeling transitions that enter state q, that is, those for which there exists a state q' with $\delta_i(q', \sigma/\tau, q)$.

Example 2.1 The setting $\mathcal{S}$ in Figure 1 contains three classes: HUMAN, COOK, and EGG. The set of simple requests is {*eat, prepare, boil, fry*}. In order to fulfill these requests, the class HUMAN uses control requests too. The bound on the class HUMAN is 1, and the bound on the classes COOK and EGG is 2.

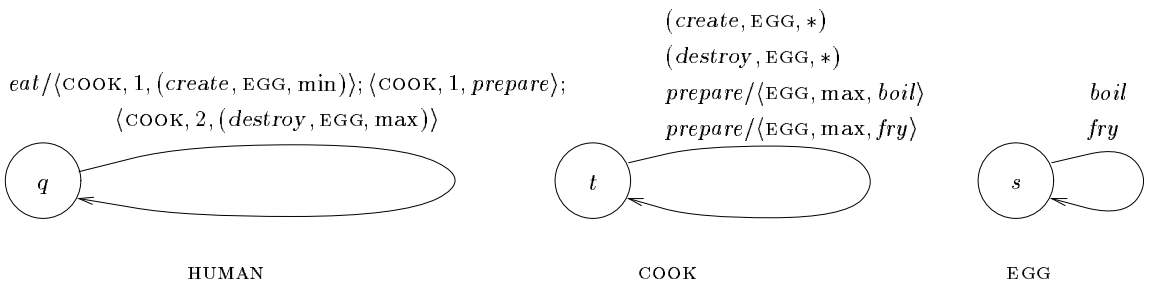

Figure 1: A setting with three classes.

We assume a system S that initially contains one instance of HUMAN, two instances of COOK, and none of EGG. Note that while the classes HUMAN and EGG are deterministic, the class cook is nondeterministic. Indeed, the cook has two choices for fulfilling a *prepare* request. □

2.2 System activation

A system is activated by some external directed simple request that is directed to one of the system's components. For example, the directed request $\langle \text{HUMAN}, 1, eat \rangle$ activates the system

S in Example 2.1. When a system is active, each of its components may be either *suspended* or *attentive*. A component is suspended when it makes a transition from a certain state to another state, which may involve waiting for the action of the transition's guard to be accomplished. Otherwise, the component is attentive.

We describe the *status* of a component $O_i[x]$ by a word

$$\rho = \tau \mapsto q,$$

where $q \in Q_i$ is the target state of $O_i[x]$, and $\tau \in (\mathcal{O} \times R \times \Sigma)^*$ is the action that should be accomplished in order for $O_i[x]$ to move to q. Thus a status tells us what is still expected to happen in the future in the instance in question. Note that a component is attentive (in state q) iff τ is empty. We use Θ to denote the set of all possible statuses. We sometimes refer to τ by $action(\rho)$ and to q by $state(\rho)$.

With each directed request that is not yet fulfilled we associate the component that has requested it and a *mode* $m \in \{w, \mathrm{a}\}$. Intuitively, the request is in mode w (waiting) if the system has not started yet to take care of it, and is in mode a (active) otherwise. A *full request* is a 4-tuple $[O_j[x], O_i[y], \sigma, m]$, denoting a directed request $\langle O_i, y, \sigma \rangle$, with mode m, requested by $O_j[x]$. When the directed request is external, we replace $O_j[x]$ with *env*, to indicate that the environment has initiated the request.

2.3 Positions and configurations

We now define what the system looks like when 'frozen' at a given point in its dynamic behavior.

A *position* of S is a partial function

$$p : \mathcal{O} \times \mathbb{N}^+ \to \Theta,$$

providing a class and an index with a status. The function p is required to be defined only for pairs $\langle O, x \rangle$ for which $O[x]$ is an existing component in S.

When all the components in a position p are attentive, we say that p is *stable*. For example, the function p_1 with $p_1(\textsc{human}, 1) =\mapsto q$, $p_1(\textsc{cook}, 1) =\mapsto t$, and $p_1(\textsc{cook}, 2) =\mapsto t$ is a possible stable position of the system S from Example 2.1. When some of the components in a position p are suspended, we say that p is *unstable*. For example, the function p_2 with $p_2(\textsc{human}, 1) = \langle \textsc{cook}, 1, \mathit{prepare} \rangle; \langle \textsc{cook}, 2, (\mathit{destroy}, \textsc{egg}, \mathit{max}) \rangle \mapsto q$, $p_2(\textsc{cook}, 1) = p_2(\textsc{cook}, 2) =\mapsto t$, and $p_2(\textsc{egg}, 1) =\mapsto s$ is a possible unstable position of S. Here, the components $\textsc{egg}[1]$, $\textsc{cook}[1]$, and $\textsc{cook}[2]$ are attentive, and the component $\textsc{human}[1]$ is suspended: it is waiting for the directed requests $\langle \textsc{cook}, 1, \mathit{prepare} \rangle$ and $\langle \textsc{cook}, 2, (\mathit{destroy}, \textsc{egg}, \mathit{max}) \rangle$ to be fulfilled. Upon their fulfillment, $\textsc{human}[1]$ would move to q. In every position, there may be at most β_i instances of the class O_i. Formally, $p(O_i, j)$ is defined for at most β_i integers j.

Each position contains information about the instances of each class that are currently 'alive', i.e., they exist in the system at the present moment. It also indicates whether these

instances are attentive or suspended. This information enables a straightforward evaluation of references in R. We distinguish between two types of evaluations, represented by the partial functions

$$e\_eval_p : \mathcal{O} \times R \to \mathbb{N}^+ \qquad \text{and} \qquad a\_eval_p : \mathcal{O} \times R \to \mathbb{N}^+.$$

While e_eval_p only checks which instances of each class exist, a_eval_p also checks their availability, namely whether they are attentive. Formally, for a position p and a class O, we define the sets of indices

$$e\_comp_p(O) \qquad \text{and} \qquad a\_comp_p(O)$$

by letting $e\_comp_p(O)$ denote those indices x such that $O[x]$ is a component of S in p (i.e., $p(O,x)$ is defined), and letting $a\_comp_p(O)$ denote those indices x such that $O[x]$ is an attentive component of S in p (i.e., $p(O,x) =\!\!\mapsto s$ for some s). For every class O, we capture the true index pointed at by the reference η by defining the functions e_eval_p and a_eval_p as follows.

- If $\eta \in \mathbb{N}^+$, then $e\_eval_p(O,\eta) = a\_eval_p(O,\eta) = \eta$.
- If $\eta = min$, then $e\_eval_p(O,\eta) = min e\_comp_p(O)$ and $a\_eval_p(O,\eta) = min a\_comp_p(O)$.
- If $\eta = max$, then $e\_eval_p(O,\eta) = max e\_comp_p(O)$ and $a\_eval_p(O,\eta) = max a\_comp_p(O)$.
- In the last two cases, if $e\_comp_p(O)$ is empty, then $e\_eval_p(O,\eta)$ is undefined, and similarly, if $a\_comp_p(O)$ is empty, then $a\_eval_p(O,\eta)$ is undefined.

When we create a new instance of a class using an indirect reference, we define the index of the new instance to be the first free index for the class. Formally, given a position p, a class O, and $\eta \in R$, we define

$$new_p(O,\eta) = \begin{cases} \eta & \text{If } \eta \in \mathbb{N}^+. \\ e\_eval_p(O,\eta) + 1 & \text{If } \eta = max. \\ min\{v \in \mathbb{N}^+ : v \notin e\_comp_p(O)\} & \text{If } \eta = min. \end{cases}$$

For example, $new_{p_2}(\text{EGG}, min) = new_{p_2}(\text{EGG}, max) = 2$.

A *configuration* of S is a pair

$$c = \langle p, \Gamma \rangle,$$

where p is a position of S and Γ is a stack of full requests. Typically, Γ contains the external directed request that initiated the activity of the system, and exactly all the directed requests that appear in the actions of the statuses in p. Intuitively, these directed requests still need to be accomplished in order for S to move to a stable position. In particular, the configuration c is stable iff p is stable and Γ is empty. The order of the full requests in Γ corresponds to the order in which they should be fulfilled. Specifically, an active request is fulfilled exactly when all the requests above it in the stack have been fulfilled.

A configuration $\langle p, \Gamma \rangle$ is *initial* if $\Gamma = \langle\ \rangle$ is the empty stack, and for all O and i for which $p(O,i)$ is defined, the action $action(p(O,i))$ is empty and $state(p(O,i)) \in Q_i^0$. We assume that

a system initially has only a finite number of instances of each class (even the unbounded ones), thus $p(O, i)$ is defined for only finitely many pairs. Note that an initial configuration is stable. A configuration is *ripe* if the top element of its stack is an active full request, and it is *unripe* otherwise.

We use C to denote the set of all configurations of a given system S, and C^0 to denote the set of all its initial configurations. For example, the set $C^0 = \{\langle p_1, \langle\,\rangle\rangle\}$ contains the single initial configuration of the system S from Example 2.1.

2.4 Stack operations

We describe stacks by tuples, with the left element of the tuple being the top of the stack. In order to help us in the definitions of system dynamics below, we will be using the following functions on stacks. Let E be a set and let ST (NST) be the set of stacks (nonempty stacks, respectively) over E.

- $\mathtt{top} : NST \to E$. Given a nonempty stack Γ, the function $\mathtt{top}(\Gamma)$ returns its top element. For example,
$$\mathtt{top}(\langle e_1, e_2, \ldots, e_d\rangle) = e_1.$$

- $\mathtt{push} : ST \times E \to NST$. Given a stack Γ and an element e, the function $\mathtt{push}(\Gamma, e)$ returns the stack obtained from Γ by pushing on it the element e. For example,
$$\mathtt{push}(\langle e_1, e_2, \ldots, e_d\rangle, e) = \langle e, e_1, e_2, \ldots, e_d\rangle.$$

- $\mathtt{append} : ST \times ST \to ST$. Given given two stacks Γ_1 and Γ_2, the function $\mathtt{append}(\Gamma_1, \Gamma_2)$ returns the stack obtained by placing the stack Γ_1 on the top of the stack Γ_2. For example,
$$\mathtt{append}(\langle e_1, e_2, \ldots, e_d\rangle, \langle e'_1, e'_2, \ldots, e'_{d'}\rangle) = \langle e_1, e_2, \ldots, e_d, e'_1, e'_2, \ldots, e'_{d'}\rangle.$$

- $\mathtt{pop} : NST \to ST$. Given a nonempty stack Γ, the function $\mathtt{pop}(\Gamma)$ returns the stack obtained from Γ by popping out the top element of Γ. For example,
$$\mathtt{pop}(\langle e_1, e_2, \ldots, e_d\rangle) = \langle e_2, \ldots, e_d\rangle.$$

Finally, for a component $O[x]$ and an action τ, the stack $\mathtt{compose}(O[x], \tau)$ of full requests is obtained from τ by replacing each directed request $\langle O', \eta, \sigma\rangle$ in τ by the full request $[O[x], O'[\eta], \sigma, w]$. For example,

$$\mathtt{compose}(O_1[1], \langle O_2, \mathit{min}, \sigma_1\rangle\,;\,\langle O_3, \mathit{max}, \sigma_3\rangle\,;\,\langle O_2, 2, \sigma_3\rangle) =$$
$$\langle [O_1[1], O_2[\mathit{min}], \sigma_1, w], [O_1[1], O_3[\mathit{max}], \sigma_3, w], [O_1[1], O_2[2], \sigma_3, w]\rangle.$$

2.5 System reaction

We now define how one configuration leads to another in the system's dynamic behavior, and for this we must first define deadlocks. We say that c is a *deadlock* configuration if one of the following holds:

- The first request that needs to be fulfilled in c is directed to a nonexisting or suspended instance. Formally, $c = \langle p, \Gamma \rangle$ is such that $\texttt{top}(\Gamma) \in [\_, O[\eta], \_, \_]$, and either $\text{a\_}eval_p(O, \eta)$ is undefined, or $\text{a\_}eval_p(O, \eta) = x$ and Γ contains a full request in $[O[x], \_, \_, \_]$. Note that the latter case is possible only if $\eta \in \mathbb{N}^+$.
- The first request that needs to be fulfilled in c is a destruction request directed to a suspended component. Formally, $c = \langle p, \Gamma \rangle$ is such that $\texttt{top}(\Gamma) \in [\_, \_, (destroy, O, \eta), \text{a}]$, and the component $O[\text{e\_}eval_p(O, \eta)]$ is suspended.
- The first request that needs to be fulfilled in c is a creation request to a class O_i that has already reached its allowed bound β_i of instances. Formally, $c = \langle p, \Gamma \rangle$ is such that $\texttt{top}(\Gamma) \in [\_, \_, (create, O, \eta), \text{a}]$, the bound for O is β, and $new_p(O, \eta) > \beta$.

Given two configurations $c = \langle p, \Gamma \rangle$ and $c' = \langle p', \Gamma' \rangle$, we say that c' is a *successor of c in S* if c is not a deadlock configuration and one of the following holds.

1. The configuration c is stable, with $\Gamma = \langle\ \rangle$, and there exist a directed simple request $\langle O_i, \eta, \sigma \rangle$ and a transition $\delta_i(state(p(O_i, x)), \sigma/\tau, q)$, for $x = \text{a\_}eval_p(O_i, \eta)$, such that the following hold:
   - $p'(O_i, x) = \tau \mapsto q$, and for all other pairs $\langle O, z \rangle \in \mathcal{O} \times \mathbb{N}^+$, we have $p'(O, z) = p(O, z)$.
   - $\Gamma' = \texttt{append}(\texttt{compose}(O_i[x], \tau), [env, O_i[x], \sigma, \text{a}])$.
2. The configuration c is unstable, and $\texttt{top}(\Gamma) \in [\_, \_, \_, w]$. Let $\texttt{top}(\Gamma) = [O_j[y], O_i[\eta], \sigma, w]$. Then, there exists a transition $\delta_i(state(p(O_i, x)), \sigma/\tau, q)$, for $x = \text{a\_}eval_p(O_i, \eta)$, such that the following hold:
   - $p'(O_i, x) = \tau \mapsto q$, and for all other pairs $\langle O, z \rangle \in \mathcal{O} \times \mathbb{N}^+$, we have $p'(O, z) = p(O, z)$.
   - $\Gamma' = \texttt{append}(\texttt{compose}(O_i[x], \tau), \texttt{push}([O_j[y], O_i[x], \sigma, \text{a}], \texttt{pop}(\Gamma)))$.
3. The configuration c is unstable, and $\texttt{top}(\Gamma) \in [\mathcal{O}[\_], \_, \_, \text{a}]$. Let $\texttt{top}(\Gamma) = [O_j[y], O_i[x], \sigma, \text{a}]$. Then, it must be that $p(O_j, y) = \langle O_i, \eta, \sigma \rangle\ ; \tau \mapsto q$ for some η, τ, and q, and
   - If $\sigma \in \Sigma$, then $p'(O_j, y) = \tau \mapsto q$, and for all other pairs $\langle O, z \rangle \in \mathcal{O} \times \mathbb{N}^+$, we have $p'(O, z) = p(O, z)$.
   - If $\sigma = (destroy, O_k, \eta)$, then $p'(O_j, y) = \tau \mapsto q$, $p'(O_k, \text{e\_}eval_p(O_k, \eta))$ is undefined, and for all other pairs $\langle O, z \rangle \in \mathcal{O} \times \mathbb{N}^+$, we have $p'(O, z) = p(O, z)$.

- If $\sigma = (create, O_k, \eta)$, then $p'(O_j, y) = \tau \mapsto q$, $p'(O_k, new_p(O_k, \eta)) = \mapsto q_k^0$, for some $q_k^0 \in Q_k^0$, and for all other pairs $\langle O, z\rangle \in \mathcal{O} \times \mathbb{N}^+$, we have $p'(O, z) = p(O, z)$.
- $\Gamma' = \mathtt{pop}(\Gamma)$.

4. The configuration c is unstable, and $\Gamma = \langle [env, O_i[x], \sigma, \mathrm{a}]\rangle$ for some $O_i[x]$ and σ. Then, it must be the case that p is stable, $\sigma \in \Sigma$, $p' = p$, and $\Gamma' = \langle\ \rangle$.

The various successor relations described above can be understood as follows, where the first two cases involve unripe configurations and the other two involve ripe ones.

In cases 1 and 2, $c = \langle p, \Gamma\rangle$ is an unripe configuration. Moving from c to its successor c' corresponds to the system deciding how to fulfill a directed request and preparing for it to be fulfilled, but not actually fulfilling it yet. When c is stable, the directed request is external, in which case it must be simple. When c is unstable, the directed request is the one waiting at the top of the stack. In both cases, the direct reference x to the instance of the class O_i to which the request is directed is calculated according to the current position. Thus, an indirect reference in a directed request becomes direct when the request is activated. Note, however, that if the request is a control request involving an indirect reference, then this reference is not yet calculated. As to the new position in cases 1 and 2, p' is obtained from p by changing the status of the component $O_i[x]$ to which the request is directed. The new status of $O_i[x]$ depends on the transition along which it chooses to proceed in order to fulfill the request. The new status consists of the action τ associated with this transition's guard and of its target state. When the directed request, say $\langle O_i, x, \sigma\rangle$, is external, Γ' contains the full request $[env, O_i[x], \sigma, \mathrm{a}]$. This full request indicates that $\langle O_i, x, \sigma\rangle$ was requested by the environment, and that the system is now actively taking care of it. It also contains a direct reference to the instance of O_i that is taking care of σ. Since accomplishing $O_i[x]$'s action is required in order for the system to fulfill σ, the stack Γ' also contains the directed requests in the action τ, with indication that they were requested by $O_i[x]$ and that they are still waiting for the system to take care of them. When the directed request $\langle O_i, x, \sigma\rangle$ is not external, Γ' is updated by making the reference to the instance $O_i[x]$ direct, changing the mode of the request at the top of the stack to a and, as above, by also containing the directed requests whose accomplishment is required in order to fulfill σ by $O_i[x]$.

In both these cases, 1 and 2, we say that c' is a *silent successor* of c in S and write $s\_succ_S(c, c')$.

In cases 3 and 4, $c = \langle p, \Gamma\rangle$ is a ripe configuration, and moving from c to its successor c' corresponds to the system actually fulfilling a directed request. In both cases, the directed request, say $\langle O_i, x, \sigma\rangle$, is active and is at the top of the stack. If p is stable, the directed request is external, and we only need pop it off the stack (and move to a stable configuration). Since an external request must be simple, this involves no changes in the position of the system. If p is unstable and the directed request was requested by, say, object $O_j[y]$, we also need to update the position of $O_j[y]$ by removing the request $\langle O_i, \eta, \sigma\rangle$ (for the corresponding $\eta \in R$) from its status. Moreover, if the request is a control request, this may involve the destruction or

creation of some new instance. Since c is not a deadlock configuration, destruction requests are directed either at an instance that does not exists (in which case no destruction takes place) or at an attentive instance. Also, creation requests are directed at a class that has no more instances than its bound. Then, the creation results in a new instance or an initialization of an existing one.

In these cases, 3 and 4, when $\langle O_i, x, \sigma \rangle$ is simple, we say that c' is a $\langle O_i, \sigma \rangle$-*successor* of c in S and write $succ_S(c, O_i.\sigma, c')$. When $\langle O_i, x, \sigma \rangle$ is a control request, we say, again, that c' is a *silent successor* of c in S and write $s\_succ_S(c, c')$.

Note that while an unripe configuration may have many silent successors, a ripe configuration has only a single successor. This unique successor is silent if the directed request at the top of the stack is a control request, and it is not silent if the request is a simple one. Note also that the $succ_S$ relation ignores the particular instance that has fulfilled the requests and retains its class only.

2.6 Reachability and system runs

For two configurations c and c', we say that c' is $\langle O_i, \sigma \rangle$-*reachable* from c in S, and write $succ^*_S(c, O_i.\sigma, c')$, if there is a sequence $c_0, c_1, \ldots, c_m$ of configurations such that $c_0 = c$, $c_m = c'$, $succ_S(c_{m-1}, O_i.\sigma, c_m)$, and for all $1 \leq i < m$, we have $s\_succ_S(c_{i-1}, c_i)$. Thus, c' is $\langle O_i, \sigma \rangle$-reachable from c in S iff there is a (possibly empty) sequence of silently successive configurations that lead from c to a configuration c'' such that c' is an $\langle O_i, \sigma \rangle$-successor of c''.

A configuration c is *reachable* in the system S if there exists a finite sequence of successive configurations $c_0, \ldots, c_m$, such that $c_0 \in C^0$ and $c_m = c$.

We now discuss sequences of configuration changes, constituting an entire execution, or run, of the system.

A *run* of S is a sequence $r = c_0, c_1, \ldots$ of successive configurations such that c_0 is stable, and either r is infinite, in which case c_i is unstable for all $i > 0$, or $r = c_0, \ldots, c_m$ is finite, in which case c_i is unstable for all $0 < i < m$ and c_m is either stable or a deadlock. We say that a system S is *deadlock free* if there is no reachable deadlock configuration in S. Thus, if S is a deadlock free system, all the runs of a S that start in a reachable stable configuration are either infinite or terminate in a stable configuration.

2.7 Bounded systems

The definition of runs considers both finite and infinite sequences of successive configurations. We now suggest a natural restriction on systems that guarantees the finiteness of its runs. To motivate this, here is an example of a system that admits an infinite run:

Example 2.2 The setting in Figure 2 contains two classes: O_1 and O_2. The set of simple requests is the singleton $\{a\}$. The bound on class O_1 is ∞, and the bound on class O_2 is 1.

We assume a system based on this setting, that initially contains one instance of O_1 and one instance of O_2.

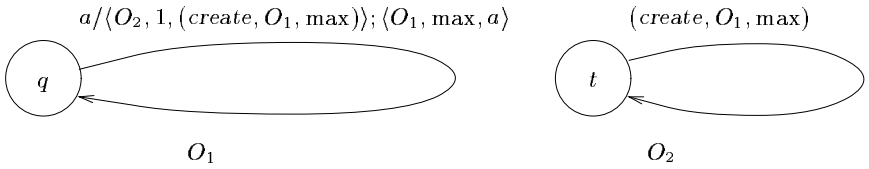

Figure 2: A setting that is a base for a system with an infinite run.

In order to fulfill the external request $\langle O_1, 1, a\rangle$, the component $O_1[1]$ requests the component $O_2[1]$ to create a new instance of the class O_1, and then requests this instance, which receives the index 2, to fulfill a. In order to fulfill the directed request $\langle O_1, 2, a\rangle$, the component $O_1[2]$ initiates the creation of another new instance of the class O_1, receiving the index 3, and the same routine continues, resulting in infinitely many instances of O_1, created during an infinite run. □

Example 2.2 illustrates an unbounded 'creation cycle' of instances, which we might want to avoid. After formally defining this notion, we show that acyclic instance creation is a sufficient condition for guaranteeing the finiteness of runs.

For two classes O and O' and two requests σ and σ', we say that O *may ask* σ' *from* O' *for* σ if there is a transition $\langle q_1, \sigma/\tau, q_2\rangle$ of O such that τ contains a directed request of the form $\langle O', \_, \sigma'\rangle$. In other words, O may fulfill σ by accomplishing an action in which some instance of O' is required to fulfill σ'.

For two classes O and O', we say that O *may create* O' if there is a finite sequence $O_1, O_2, \ldots, O_n$ of classes and a finite sequence $\sigma_1, \ldots, \sigma_n$ of requests, such that $O_1 = O$, for all $1 \leq i \leq n-1$, the class O_i may ask σ_{i+1} from O_{i+1} for σ_i, and σ_n is of the form $(create, O', \_)$. Intuitively, O may create O' if it can take a transition that may eventually lead to the creation of an instance of O'. In Example 2.2, O_1 may create O_1, but is not the case that O_2 may create O_1. Indeed, while O_2 can fulfill a requirement to create O_1, it cannot initiate such a creation.

We say that a setting $\mathcal{S}$ is *strictly bounded* if all its classes are bounded. It is said to be *bounded* if there is no unbounded class O in $\mathcal{S}$ such that O may create O. A system is strictly bounded if it is based on a strictly bounded setting, and it is bounded if it is based on a bounded setting.

Recall that each class O_i may have at most β_i instances in a system that is based on a setting in which the bound on O_i is β_i. When the setting is strictly bounded, this guarantees that every class has only finitely many instances. It can be shown that when the setting is bounded, we can also bound the number of instances of all classes by a finite number. For that, we define, for each class O_i, a *context-sensitive bound* β_i'. Unlike β_i, the bound β_i' depends on the other

classes in the setting, on the fact that the setting is bounded and on the initial configuration of the system. (Recall that the initial configuration of a system specifies the initial number of instances of each class, which must be finite.)

We can now show that a bounded system cannot have an infinite run. For that, we first bound the number of configurations that a bounded system may have. When the system is not strictly bounded, the arguments are exactly the same, with β_i' replacing β_i.

Lemma 2.3 *The number of configurations of a bounded system is finite.*

Proof: Consider a system S over a bounded setting $\mathcal{S} = \langle \Sigma, O_1, \ldots, O_n \rangle$, in which we have $O_i = \langle Q_i, Q_i^0, \delta_i, \beta_i \rangle$. We first consider a strictly bounded $\mathcal{S}$, in which case we prove that the number of configurations of S is bounded by

$$(\prod_{i=1}^{n} \beta_i)! \cdot \prod_{i=1}^{n} (\beta_i \cdot \prod_{q \in Q_i} (\prod_{\sigma/\tau \in into(q)} |\tau| + 1)).$$

In every configuration, each component $O_i[x]$ of the class O_i of $\mathcal{S}$ has a status $\rho = \tau \mapsto q$. By the definition of successive configurations, τ must be a (possibly empty) suffix of some action in $into(q)$. Accordingly, component O_i may have at most $\prod_{q \in Q_i} \prod_{\sigma/\tau \in into(q)} |\tau| + 1$ different statuses. Consider a configuration $c = \langle p, \Gamma \rangle$. Once we fix the status of each of the components in p, we also fix the set of full requests in Γ, yet we do not fix their order in Γ. Such an order is imposed by an order on the set of components. Indeed, the full requests of an object form a single 'block' in Γ. Hence the additional factor of $(\prod_{i=1}^{n} \beta_i)!$.

To see that the full requests in the stack come in blocks, assume by way of contradiction that the stack Γ of the configuration c contains two full requests r_1 and r_2 of the form $[O_i[x], \_, \_, \_]$, and a full request r_3 of the form $[O_j[y], \_, \_, \_]$, with $\langle i, x \rangle \neq \langle j, y \rangle$, between them. Without loss of generality, assume that r_2 is above r_1 in the stack. By the definition of successive configurations (cases 1 and 2), full requests are pushed onto the stack in blocks. Therefore, when a run reaches the configuration c, the run must have a prefix that ends in some configuration $c' = \langle p', \Gamma' \rangle$, such that the stack Γ' contains r_1 and r_3 but does not contain r_2, and $\mathtt{top}(\Gamma') \in [\_, O_i[x], \_, \_]$. The configuration c' is a deadlock configuration, contradicting the reachability of c in the run. □

For a system S based on a bounded setting $\mathcal{S}$, let $\|conf_{\mathcal{S}}(S)\|$ denote the bound on the number of configurations that S may have. As explained above, when some of the classes in $\mathcal{S}$ are unbounded, the finiteness of such bound may depends on the set C^0, of S's initial configurations.

Lemma 2.4 *All the runs of a bounded system are finite.*

Proof: We actually prove a stronger claim, namely that all the unstable configurations in a run are different. The lemma then follows from the fact that S has only finitely many configurations, and from the fact that all the configurations of a run, with the possible exception of two, are unstable.

Consider a run $r = c_0, c_1, \ldots$ For all indices k, let $c_k = \langle p_k, \Gamma_k \rangle$, and let d_k denote the depth of the stack Γ_k. We prove that for every depth $d \geq 0$ and no configuration $c = \langle p, \Gamma \rangle$ with stack d can lead to a configuration that agrees with Γ on the bottom d elements without visiting a stable configuration. In particular, if there are two indices $0 < i < j$ for which $c = c_i = c_j$, there must be a stable configuration c_k, for $i < k < j$, contradicting the fact that r is a run.

The proof proceeds by induction on d. First, if $d = 1$, the stack Γ is of the form $\langle [env, \_, \_, \mathrm{a}] \rangle$, and the only configuration reachable from c is stable. For the induction step, we distinguish between two cases. If $\mathtt{top}(\Gamma)$ is an active full request $[O[x], O'[y], \sigma, \mathrm{a}]$, then the successor configuration of c has a stack with $d-1$ elements, and the claim follows from the induction hypothesis. If $\mathtt{top}(\Gamma)$ is a waiting full request $[O[x], O'[y], \sigma, w]$, then the successor configuration of c agrees with Γ on all the $d-1$ bottom elements, and it has the full request $[O[x], O'[y], \sigma, \mathrm{a}]$ as its d'th element. In order to reach a configuration whose stack agrees with Γ on its bottom d elements, this full request has to be popped out. This, however, involves a stack of depth $d-1$, to which the induction hypothesis applies. □

2.8 The trace set

We now abstract away some of the unimportant details of a system's runs, to obtain the behavioral 'signature' of the system as a language over an appropriate alphabet of request symbols.

Specifically, a system based on the setting $\mathcal{S}$ will generate a language over the alphabet $A = \mathcal{O}.\Sigma$. A letter in $\mathcal{O}.\Sigma$ corresponds to a directed simple request, which means that in our definition of the language of a system we ignore control requests and transitions that move from an unripe configuration to a silent successor.

Consider a system S, a stable configuration c, and a directed simple request $\langle O, x, \sigma \rangle$. The *trace set of S from c on $\langle O, x, \sigma \rangle$* is the language $\mathcal{L}_c^{O.\sigma} \subseteq A^*$ of all words w that describe a sequence of directed simple requests that the system in configuration c may fulfill in order to accomplish the directed request $\langle O, x, \sigma \rangle$. The language ignores the particular instances of the classes that are involved in accomplishing the directed requests and refers to the classes only (that is, the alphabet is $(\mathcal{O}.\Sigma)$ rather than $\mathcal{O} \times R \times \Sigma$). The order of the directed requests in such a w corresponds to the order in which they are accomplished, so that, in fact, we have $last(w) = O.\sigma$, where $last(w)$ is the last letter in w. Formally, a word $w = w_0 \cdot w_1 \cdots w_k$ is in $\mathcal{L}_c^{O.\sigma}$ iff there exists a sequence $c_0, \ldots, c_{k+1}$ of configurations of S such that $c_0 = c$ and for all $1 \leq i \leq k$, we have $succ_S^*(c_i, w_i, c_{i+1})$. We say that $c_0, \ldots, c_{k+1}$ *witnesses* the membership of w in $\mathcal{L}_c^{O.\sigma}$.

Note that since S is nondeterministic, a word w may have several witnesses to its membership in $\mathcal{L}_c^{O.\sigma}$. For two configurations c and c', and a word $w \in A^*$, we say that w *leads from* c *to* c', denoted $leads(c, w, c')$, if there exists a witness $c, \ldots, c'$ to the membership of w in $\mathcal{L}_c^{last(w)}$.

Consider a finite or infinite sequence $\xi = \langle O^0, x^0, \sigma^0\rangle \cdot \langle O^1, x^1, \sigma^1\rangle \cdot \langle O^2, x^2, \sigma^2\rangle \cdots$ of directed simple requests. The *trace set of* S *on* ξ is the language $\mathcal{L}^\xi \subseteq (A^* \cup A^\omega)$ such that a word $w = w_0 \cdot w_1 \cdot w_2 \cdots$ is in $\mathcal{L}^\xi$ iff for each $w_i \in A^*$, we have $last(w_i) = O^i.\sigma^i$, and there exists a sequence $c_0, c_1, c_2 \ldots$ of stable configurations $c_i \in C$ (these sequences are finite iff ξ is finite), such that $c_0 \in C^0$, and for all $j \geq 0$ we have $leads(c_j, w_j, c_{j+1})$. When S is not clear from the context we denote the trace set of S on ξ by $\mathcal{L}_S^\xi$ instead of $\mathcal{L}^\xi$.

2.9 More on the egg/cook example

Consider the system S from Figure 1. The initial configuration of S is $c_0 = \langle p_0, \Gamma_0\rangle$, where $p_0(\text{HUMAN}, 1) =\!\mapsto q$, $p_0(\text{COOK}, 1) = p_0(\text{COOK}, 2) =\!\mapsto t$, and $\Gamma_0 = \langle\,\rangle$. The trace set of S from c_0 on the directed request $\langle \text{HUMAN}, 1, eat\rangle$ contains the finite word $w = \text{EGG}.boil; \text{COOK}.prepare; \text{HUMAN}.eat$. A witness to the membership of w in $\mathcal{L}_{c_0}^{\text{HUMAN}.eat}$ is described below. We first describe the relevant run of S. Each configuration $c_i = \langle p_i, \Gamma_i\rangle$ is described by p_i and Γ_i. When we describe p_{i+1} we show only pairs in $\mathcal{O} \times \mathbb{N}$ for which $p_{i+1} \neq p_i$. In particular, $p_{i+1}(O, x) = \perp$ means that although in p_i the instance $O[x]$ exists, it no longer exists in p_{i+1} (that is, $p_{i+1}(O, x)$ is undefined).

- $p_0(\text{HUMAN}, 1) =\!\mapsto q$.
- $p_0(\text{COOK}, 1) = p_0(\text{COOK}, 2) =\!\mapsto t$.
- $\Gamma_0 = \langle\,\rangle$.

  The initial configuration has one instance of HUMAN and two instances of COOK. The configuration is stable with an empty stack. Hence the system is ready to handle external requests from the environment. Among the cases described in Section 2.5, this corresponds to case 1. The request $\langle \text{HUMAN}, 1, eat\rangle$ eventually arrives, inducing the full request $[env, \text{HUMAN}[1], eat, \text{a}]$. In order to fulfill the simple request *eat*, the class HUMAN asks the first cook to create an egg and prepare it, and then asks the second cook to destroy the egg. Accordingly, we have:

- $p_1(\text{HUMAN}, 1) = \langle \text{COOK}, 1, (create, \text{EGG}, min)\rangle; \langle \text{COOK}, 1, prepare\rangle;$
  $\langle \text{COOK}, 2, (destroy, \text{EGG}, max)\rangle \mapsto q$.
- $\Gamma_1 = \langle [\text{HUMAN}[1], \text{COOK}[1], (create, \text{EGG}, min), \text{w}], [\text{HUMAN}[1], \text{COOK}[1], prepare, \text{w}],$
  $[\text{HUMAN}[1], \text{COOK}[2], (destroy, \text{EGG}, max), \text{w}], [env, \text{HUMAN}[1], eat, \text{a}]\rangle$.

  The configuration c_1 is unstable and unripe. This corresponds to case 2. The component COOK[1] takes the transition guarded by $(create, \text{EGG}, 1)$ in order to fulfill the control

request at the top of the stack. Since the task of this transition is empty, no new requests are pushed onto the stack, and the position of COOK[1] does not change:

- $p_2 = p_1$.
- $\Gamma_2 = \langle[\text{HUMAN}[1], \text{COOK}[1], (\mathit{create}, \text{EGG}, \mathit{min}), \text{a}], [\text{HUMAN}[1], \text{COOK}[1], \mathit{prepare}, \text{w}],$
  $[\text{HUMAN}[1], \text{COOK}[2], (\mathit{destroy}, \text{EGG}, \mathit{max}), w], [\mathit{env}, \text{HUMAN}[1], \mathit{eat}, \text{a}]\rangle$.

  The configuration c_2 is ripe. This corresponds to case 3, and the system fulfills the request at the top of the stack. This involves an update of the position of HUMAN[1], which requested it, and a creation of a new instance of EGG:
- $p_3(\text{HUMAN}, 1) = \langle \text{COOK}, 1, \mathit{prepare}\rangle; \langle \text{COOK}, 2, (\mathit{destroy}, \text{EGG}, \mathit{max})\rangle \mapsto q$.
- $p_3(\text{EGG}, 1) = \!\!\mapsto s$.
- $\Gamma_3 = \langle[\text{HUMAN}[1], \text{COOK}[1], \mathit{prepare}, w], [\text{HUMAN}[1], \text{COOK}[2], (\mathit{destroy}, \text{EGG}, \mathit{max}), w],$
  $[\mathit{env}, \text{HUMAN}[1], \mathit{eat}, \text{a}]\rangle$.

  Like c_1, the configuration c_3 is unstable and unripe, corresponding to case 2. The component COOK[1] takes the transition guarded by *preapare* in order to fulfill the simple request at the top of the stack. Since the task of this transition is not empty, a new request is pushed onto the stack, and COOK[1] becomes suspended:
- $p_4(\text{COOK}, 1) = \langle \text{EGG}, 1, \mathit{boil}\rangle \mapsto t$.
- $\Gamma_4 = \langle[\text{COOK}[1], \text{EGG}[1], \mathit{boil}, w], [\text{HUMAN}[1], \text{COOK}[1], \mathit{prepare}, \text{a}],$
  $[\text{HUMAN}[1], \text{COOK}[2], (\mathit{destroy}, \text{EGG}, \mathit{max}), w], [\mathit{env}, \text{HUMAN}[1], \mathit{eat}, \text{a}]\rangle$.

  The configuration c_4 is also unstable and unripe. The component EGG[1] takes the transition guarded by *boil* in order to fulfill the simple request at the top of the stack. Since the task of this transition is empty, no new requests are pushed onto the stack, and the position of EGG[1] does not change:
- $p_5 = p_4$.
- $\Gamma_5 = \langle[\text{COOK}[1], \text{EGG}[1], \mathit{boil}, \text{a}], [\text{HUMAN}[1], \text{COOK}[1], \mathit{prepare}, \text{a}],$
  $[\text{HUMAN}[1], \text{COOK}[2], (\mathit{destroy}, \text{EGG}, \mathit{max}), w], [\mathit{env}, \text{HUMAN}[1], \mathit{eat}, \text{a}]\rangle$.

  The configuration c_5 is ripe, and the system fulfills the request at the top of the stack. This involves an update of the position of COOK[1], which requested it.
- $p_6(\text{COOK}, 1) = \!\!\mapsto t$.
- $\Gamma_6 = \langle[\text{HUMAN}[1], \text{COOK}[1], \mathit{prepare}, \text{a}], [\text{HUMAN}[1], \text{COOK}[2], (\mathit{destroy}, \text{EGG}, \mathit{max}), w],$
  $[\mathit{env}, \text{HUMAN}[1], \mathit{eat}, \text{a}]\rangle$.

  The configuration c_6 is again ripe:

- $p_7(\text{HUMAN}, 1) = \langle \text{COOK}, 2, (\mathit{destroy}, \text{EGG}, \mathit{max}) \rangle \mapsto q$.
- $\Gamma_7 = \langle [\text{HUMAN}[1], \text{COOK}[2], (\mathit{destroy}, \text{EGG}, \mathit{max}), w], [\mathit{env}, \text{HUMAN}[1], \mathit{eat}, \text{a}] \rangle$.

  The configuration c_7 is similar to configuration c_1, only that here HUMAN[1] waits for the second cook to destroy the egg. The component COOK[2] proceeds with the transition guarded by $(\mathit{destroy}, \text{EGG}, 1)$:
- $p_8 = p_7$
- $\Gamma_8 = \langle [\text{HUMAN}[1], \text{COOK}[2], (\mathit{destroy}, \text{EGG}, \mathit{max}), \text{a}], [\mathit{env}, \text{HUMAN}[1], \mathit{eat}, \text{a}] \rangle$.

  The configuration c_8 is ripe, and all the requests in it are active. It is left to update the positions of the involved components, while popping the requests from the stack:
- $p_9(\text{EGG}, 1) = \bot$.
- $p_9(\text{HUMAN}, 1) = \!\!\mapsto q$.
- $\Gamma_9 = \langle [\mathit{env}, \text{HUMAN}[1], \mathit{eat}, \text{a}] \rangle$.

  The configuration c_9 corresponds to case 4. The only request in the stack is active, and is an external one:
- $p_{10} = p_9$.
- $\Gamma_{10} = \langle\ \rangle$.

The run $c_0, \ldots, c_{10}$ implies that the following succession relationships hold: $\mathit{succ}^*_S(c_0, \text{EGG}.\mathit{boil}, c_6)$, $\mathit{succ}^*_S(c_6, \text{COOK}.\mathit{prepare}, c_7)$, and $\mathit{succ}^*_S(c_7, \text{HUMAN}.\mathit{eat}, c_{10})$. Thus, c_0, c_6, c_7, c_{10} witnesses the membership of

$$\text{EGG}.\mathit{boil}; \text{COOK}.\mathit{prepare}; \text{HUMAN}.\mathit{eat}$$

in $\mathcal{L}_{c_0}^{\text{HUMAN}.\mathit{eat}}$. Since the cook can satisfy the *prepare* request by fulfilling two nondeterministic choices (either *boil* or *fry* the EGG), the trace set $\mathcal{L}_{c_0}^{\text{HUMAN}.\mathit{eat}}$ contains also the word

$$\text{EGG}.\mathit{fry}; \text{COOK}.\mathit{prepare}; \text{HUMAN}.\mathit{eat}.$$

3 Substitutability: Refinement Relations

We are now finally ready to define the notions of behavioral inheritance we are interested in. Our main interest is in capturing substitutability; namely, the ability to replace an object A with another object B without losing behaviors. Thus, B inherits from A. We do so in two ways, linear and branching, in line with the classical dichotomy in the literature on concurrency and verification (see e.g., [Pnu85]). The study of refinement relations in these areas is motivated by the need to check that a given system is a correct implementation of a

given, more abstract, specification. There, we want every behavior of the implementation to be also a possible behavior of the specification. Accordingly, we check that the replacement of the specification by the implementation does not introduce new behaviors. In the case of inheritance and substitutability, we use refinement relations for the other direction: we check that the replacement of class A by class B, i.e., B inheriting from A, does not involve loss of behaviors. In Section 5, we discuss other relations between the inheriting and inherited class, and how our definitions and results here extend to them.

Consider two settings $\mathcal{S}_1$ and $\mathcal{S}_2$ over the same set Σ of simple requests. Let $\mathcal{O}_1$ and $\mathcal{O}_2$ be the set of classes in $\mathcal{S}_1$ and $\mathcal{S}_2$, respectively, and let $A_1 = \mathcal{O}_1.\Sigma$ and $A_2 = \mathcal{O}_2.\Sigma$ be the corresponding alphabets. A *refinement mapping* is a function $f : \mathcal{O}_1 \to \mathcal{O}_2$. We extend the mapping f to letters, words, and languages in A_1 in a straightforward way. Thus, f maps a letter $O.\sigma$ in A_1 to the letter $f(O).\sigma$ in A_2, it maps a word $w \in (A_1^* \cup A_1^\omega)$ to the word obtained from w by applying f to its letters, and it maps a language $\mathcal{L}$ of words over A_1 to the language of words over A_2 obtained from $\mathcal{L}$ by applying f to its words.

3.1 The linear case: trace containment

Here we want to capture the simple notion whereby (the essential part of) any run in the parent class A is also a run in the inheriting class B. This notion will thus involve single sequences; hence the term 'linear'.

We say that a system S_1 that is based on $\mathcal{S}_1$ is *trace contained* in a system S_2 that is based on $\mathcal{S}_2$, denoted $S_1 \subseteq S_2$, if there exists a refinement mapping $f : \mathcal{O}_1 \to \mathcal{O}_2$ such that for every sequence $\xi \in A_1^* \cup A_1^\omega$, we have $f(\mathcal{L}_{S_1}^{\xi}) \subseteq \mathcal{L}_{S_2}^{f(\xi)}$. That is, S_1 is trace contained in S_2 iff there exists a refinement mapping f that maps every sequence of directed requests that S_1 generates as a reaction to some input ξ_1 into a sequence of directed requests that S_2 generates as a reaction to $f(\xi)$.

When $S_1 \subseteq S_2$ with the refinement mapping f, we say that S_1 *linearly refines* S_2 *by* f. Note that since the languages generated by S_1 and S_2 ignore control requests and ignore the particular instances of a class that fulfills a request, so does our definition of linear refinement.

Linear refinement gives rise to our first notion of behavioral inheritance, which we call *linear substitutability*. A class A is linearly substitutable by B in a system S (or, if we want to use the terminology of object models and inheritance, B inherits from A, or B `is-a` A) if for some f the system S linearly refines the system obtained from S by replacing A with B, and similarly for the simultaneous substitution of a number of classes. This notion can also be defined for substituting instances rather than entire classes.

Example 3.1 Consider the setting $\mathcal{S}_1$ in Figure 3, obtained from the setting $\mathcal{S}$ of Example 2.1 by removing the transition $\langle t, \mathit{prepare}/\langle \text{EGG}, \mathit{max}, \mathit{boil}\rangle, t\rangle$ from the class COOK, and removing the transition $\langle s, \mathit{boil}, s\rangle$ from the class EGG.

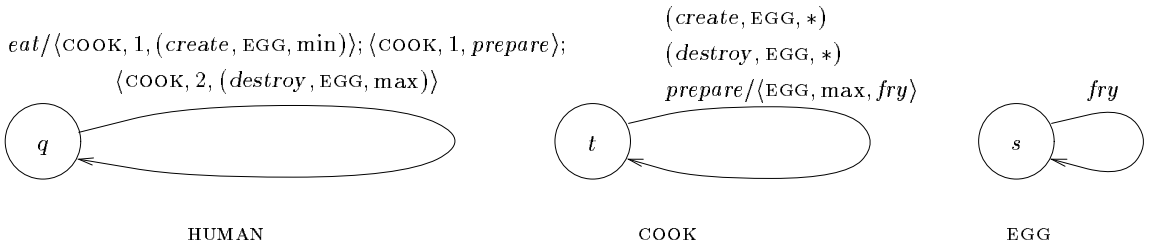

Figure 3: A non-boiling setting.

Consider a system S_1 that is based on $\mathcal{S}_1$, and which initially contains one instance of HUMAN and two instances of COOK. It is not hard to see that the system S_1 is trace contained in the system S from Example 2.1, by the identity refinement mapping. On the other hand, S is not trace contained in S_1. Indeed, while every trace of S_1 can be generated by S, whose cook has more nondeterministic choices as to how to fulfill a *prepare* request, the traces of S that follow the boiling option are impossible in S_1. It follows that the classes COOK and EGG of S_1 are linearly substitutable by these of S, but not vice versa. □

3.2 The branching case: simulation

Here we want to capture a more subtle concept of substitutability, whereby (the essential part of) any behavior in the parent class A is somehow present in the total behavior of the inheriting class B. This notion will implicitly involve trees; hence the term 'branching'.

A pair $\langle f, H\rangle$ consisting of a refinement mapping $f : \mathcal{O}_1 \to \mathcal{O}_2$ and a relation $H \subseteq C_1 \times C_2$ between the configurations of S_1 and S_2, is a *simulation pair* from S_1 to S_2 if for every c_1 and c_2 with $H(c_1, c_2)$, and for every configuration $c_1' \in C_1$ and simple directed request $O_1.\sigma \in \mathcal{O}_1.\Sigma$ such that $succ^*_{S_1}(c_1, O_1.\sigma, c_1')$, there exist a configuration $c_2' \in C_2$ such that $succ^*_{S_2}(c_2, f(O_1).\sigma, c_2')$ and $H(c_1', c_2')$.

We say that S_1 is *simulated by* S_2, denoted $S_1 \leq S_2$, if there exists a simulation pair $\langle f, H\rangle$ from S_1 to S_2 such that for every initial configuration $c_1^0 \in C_1^0$, there exists an initial configuration $c_2^0 \in C_2^0$ with $H(c_1^0, c_2^0)$. When $S_1 \leq S_2$ with refinement mapping f, we say that S_1 *branchingly refines* S_2 *by* f.

Intuitively, S_1 branchingly refines S_2 by f if every behavior of S_1 (ignoring control requests and ignoring the particular instances of a class that fulfill a request) is present also in S_2.

Branching refinement gives rise to the second notion of behavioral inheritance, *branching substitutability*. A class A is branchingly substitutable by B in a system S if for some f the system S branchingly refines the system obtained from S by replacing A with B, and similarly for the simultaneous substitution of a number of classes.

Example 3.2 Consider the setting $\mathcal{S}_2$ in Figure 4, obtained from the setting $\mathcal{S}$ of Example 2.1 by replacing the class COOK by two classes, BOILING-COOK and FRYING-COOK, such that BOILING-COOK is obtained from COOK by removing the transition $\langle t, \mathit{prepare}/\langle \text{EGG}, \mathit{max}, \mathit{fry}\rangle, t\rangle$, and FRYING-COOK is obtained from COOK by removing the transition $\langle t, \mathit{prepare}/\langle \text{EGG}, \mathit{max}, \mathit{boil}\rangle, t\rangle$. In addition, the class HUMAN is modified in $\mathcal{S}_2$ so that the *eat* request can be fulfilled by two nondeterministically chosen transitions, one that activates BOILING-COOK and one that activates FRYING-COOK. Thus, while in $\mathcal{S}$ HUMAN is deterministic and COOK is nondeterministic, in $\mathcal{S}_2$ HUMAN is nondeterministic and the two cooks are deterministic.

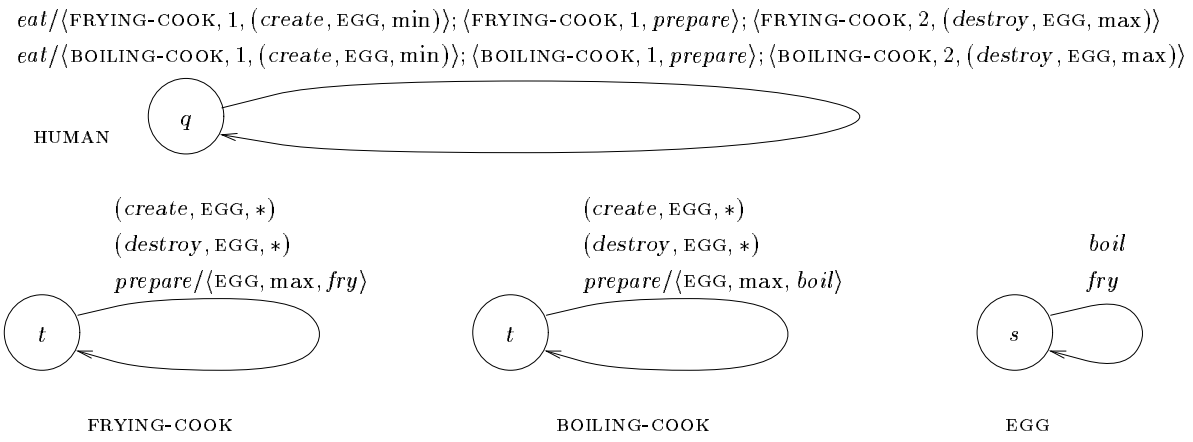

Figure 4: $S_2 \subseteq S$, yet $S_2 \not\leq S$.

Consider a system S_2 that is based on $\mathcal{S}_2$, and which initially contains one instance of each of the classes HUMAN, BOILING-COOK, and FRYING-COOK. It is not hard to see that the system S_2 is trace contained in the system S, with the refinement mapping $f(\text{HUMAN}) = \text{HUMAN}$, $f(\text{BOILING-COOK}) = f(\text{BOILING-COOK}) = \text{COOK}$, and $f(\text{EGG}) = \text{EGG}$.

On the other hand, S_2 is not simulated by S. Intuitively, in S the nondeterministic choice of boiling of frying an egg is left to the class COOK, and is therefore taken only after the *prepare* request is pushed onto the stack. In contrast, in S_2 the nondeterministic choice of boiling of frying an egg is taken by the class HUMAN, before the *prepare* request is pushed onto the stack. Another way to view this is that while the system S satisfies the specification "after the cook creates an egg he/she has the option of either boiling it or frying it", the system S_2 does not satisfy this specification. Note that the specification refers to the branching behavior of the system, namely to the possible nondeterministic choices its components may or may not take. Since the system S_2 is trace contained in S, all its linear behaviors are allowed by S. In particular, in both systems, the cook boils or fries an egg after creating it. It follows that the classes HUMAN, BOILING-COOK and FRYING-COOK of S_2 are linearly substitutable by those of S, but they are not branchingly substitutable by them. □

3.3 Branching is stronger

Recall that trace containment refers to the trace sets of systems, whereas simulation refers also to their branching structures, namely to the trees induced by systems. The ability to embed trees of one system in trees of another system is at least as hard as the ability to embed traces. Specifically, Milner showed that for systems given by state-transition graphs, simulation implies trace containment but not vice versa [Mil71]. As illustrated in Example 3.2, this is true for our object systems too:

Proposition 3.3 *Branching substitutability implies linear substitutability, but not vice versa.*

We note, however, that as with state-transition graphs, when all the components of the systems are instances of deterministic classes, simulation and trace containment coincide.

4 Detecting substitutability

Both refinement relations, trace containment and simulation, are witnessed by a refinement mapping. Usually, the refinement mapping is known, and one wants to check whether it indeed witnesses a linear or a branching refinement. In simpler words, we make a decision about which classes in one system we want to be replaced by, or to correspond to, classes in the other, and we can then talk about one system inheriting behavior from the other, i.e., substituting one for the other. In this section, we address the algorithmic/computational problem of detecting whether indeed the inheritance proposed by the mapping is behaviorally correct. That is, we want to detect substitutability.

Technically, we have to solve the following problem:

Problem 4.1 *Given two systems S_1 and S_2 and a refinement mapping $f : \mathcal{O}_1 \to \mathcal{O}_2$, does f witness a linear or a branching refinement of S_2 by S_1?*

We first define labeled state transition graphs and refinement relations for them (see, e.g., [Pnu85]).

A *labeled state transition graph* (*graph*, for short) is a tuple $G = \langle A, V, E, V^0 \rangle$, where A is an alphabet, V is a finite set of states, $E \subseteq V \times A \times V$ is a set of directed edges, labeled by letters from A, and $V^0 \subseteq V$ is a set of initial states. A *path* in G is a (finite or infinite) sequence $\pi = v_0, a_0, v_1, a_1, \ldots$ satisfying $v_0 \in V^0$ and $E(v_i, a_i, v_{i+1})$, for all $i \geq 0$. The path π induces the *trace* $\pi_{|A} = a_0 \cdot a_1 \cdots$ in $A^* \cup A^\omega$.

Consider two graphs $G_1 = \langle A, V_1, E_1, V_1^0 \rangle$ and $G_2 = \langle A, V_2, E_2, V_2^0 \rangle$ over the same alphabet. We say that G_1 is *contained in* G_2 (denoted $G_1 \subseteq G_2$) if every trace of G_1 is also a trace of G_2. A relation $H \subseteq V_1 \times V_2$ is a *simulation relation* from G_1 to G_2 if for all v_1 and v_2 with $H(v_1, v_2)$, and for all $v_1' \in V_1$ and $a \in A$ such that $E_1(v_1, a, v_1')$, there exists $v_2' \in V_2$ such that

$E(v_2, a, v_2')$ and $H(u_1', u_2')$. A simulation relation H is a *simulation* from G_1 to G_2 if for every $v_1 \in V_1^0$ there exists $v_2 \in V_2^0$ such that $H(v_1, v_2)$. If there exists a simulation from S to S', we say that G_1 *simulates* G_2 and we write $G_1 \leq G_2$.

The time complexity of checking trace containment and simulation between two graphs is well known. The trace containment problem is PSPACE-complete and the simulation problem is PTIME-complete [Mil80, SVW87, KV98].

Getting back to our subject matter, an object system can be seen to induce a labeled state transition graph, by taking the states of the graph to be the reachable configurations of the system, and its labeled transitions to be the relation *succ*.

Formally, we set this up as follows. Consider a system S over a setting $\mathcal{S} = \langle \Sigma, O_1, \ldots, O_n \rangle$ with a set C of reachable configurations. By Lemma 2.3, C is finite. The graph induced by S is then taken to be

$$G_S = \langle \mathcal{O}.\Sigma, C, succ_S, C^0 \rangle.$$

We refer to G_S as a *flat system*. Given a flat system with classes $\mathcal{O}$ and a refinement mapping $f : \mathcal{O} \to \mathcal{O}'$, the new flat system $f(G_S)$ is obtained from G_S simply by applying f to the labeling function of G_S.

Proposition 4.2 *Given two systems S_1 and S_2 and a refinement mapping $f : \mathcal{O}_1 \to \mathcal{O}_2$,*

(1) *S_1 linearly refines S_2 by f iff $f(G_{S_1}) \subseteq G_{S_2}$.*

(2) *S_1 branchingly refines S_2 by f iff $f(G_{S_1}) \leq G_{S_2}$.*

The complexity of checking substitutability now follows from the known bounds for checking trace containment and simulation stated above, along with the blowup involved in going from a system S based on a setting $\mathcal{S}$ to its flat system G_S. By the definition of G_S, this blowup is a function of the bound $\|conf_{\mathcal{S}}(S)\|$ on the number of configurations in S, and as detailed in Lemma 2.3, it may be very significant. In fact, under certain assumptions, the blowup may be even doubly-exponential: if we assume that the bounds β_i are given in binary, then a class with a description of length $O(m)$ may have 2^m instances. So if the class has 2 states, it contributes a factor of 2^{2^m} to the size of a flat system, giving rise to a double-exponential blowup even for a single class with two states. In the theorem below, we assume that the bounds β_i are either constants or are given in unary. Then, $\|conf_{\mathcal{S}}(S)\|$ is at most exponential in the description of $\mathcal{S}$, implying an exponential increase in the complexity of the trace-containment and simulation problems with respect to the complexity of the same problems for graphs. Using the techniques of [HKV97], one can show that this exponential penalty cannot be avoided in the worst case.

This discussion thus gives rise to the following:

Theorem 4.3 *Consider two systems S_1 and S_2, based on settings $\mathcal{S}_1$ and $\mathcal{S}_2$, respectively, and a refinement mapping $f : \mathcal{O}_1 \to \mathcal{O}_2$.*

(1) *The problem of deciding whether S_1 linearly refines S_2 by f is EXPSPACE-complete. Specifically, it can be solved in space polynomial in $\|conf_{\mathcal{S}_1}(S_1)\| \cdot \|conf_{\mathcal{S}_2}(S_2)\|$.*

(2) *The problem of deciding whether S_1 branchingly refines S_2 by f is EXPTIME-complete. Specifically, it can be solved in time polynomial in $\|conf_{\mathcal{S}_1}(S_1)\| \cdot \|conf_{\mathcal{S}_2}(S_2)\|$.*

In practice, however, the algorithms implicit in the upper bounds often require time and space much lower than the discouraging lower bounds. In addition, it is possible to come up with conditions that guarantee lower complexity (e.g., when S_2 is deterministic or when the guards are of bounded length). Of course, coming up with more practical syntactic restrictions and efficient algorithms to go along with them is a matter for extensive further research. In this sense, our results are but a first step, as explained in the Introduction.

5 Alternatives and extensions

We now briefly discuss some of the choices made in defining object systems and substitutability, along with possible alternatives and extensions.

5.1 Object systems

Propositional vs. first-order level reasoning Our systems have simple symbols for requests. They admit no variables, no assignment statements, no parameters, etc. As a result, our action language, for example, is also very modest. In fact, we are working here within what is often called the *propositional* level of reasoning, rather than on a *first-order* level. The propositional level is basic, i.e., more abstract, than the first-order level, and indeed should be considered first. In future work more detailed and less abstract levels of reasoning should also be investigated. In general, bad news (e.g., the high computational complexity we have exhibited) will remain at least as bad when the investigation is carried out on a less abstract and more expressive level.

Single vs. multiple thread designs The semantics of our object systems assumes single-thread requests only, in the full run-to-completion sense. Thus, as long as an object is in the process of fulfilling a request it is suspended until after the request is fulfilled. Extending our model to multiple-thread designs is possible, and to do so we would have to decide upon a clear-cut concurrency model (interleaving vs. full parallelism), a particular type of memory access, perhaps a notion of fairness, etc. Of course, if we want all of this to be UML-compliant, some of these decisions will not be ours to make. Other issues and extensions that are relevant here are the possible classification of requests into synchronous and asynchronous (see, e.g., [HG97]), into those that suspend the objects fulfilling them and those that do not (this and the synchrony/asynchrony issue are not unrelated), annotating classes with upper bounds on the maximal number of threads for their attentive objects, and more.

Finite state machines vs. statecharts On obvious restriction we have put on our models is the use of simple flat finite state machines to describe the behavior of a class, although most OOAD modeling languages use the richer language of statecharts [Har87]. Many of the features in statecharts can be incorporated into our definitions without too much trouble; e.g., hierarchy, default transitions and history. Orthogonal (concurrent) components within a statechart are of course what give them their exponential succinctness [DH94], and with this one would have to be more careful, both in the definitions and in the resulting complexity bounds. Here is an example. In our definition of object systems, the behavior of an object is independent of the status of the other objects. Now, in the spirit of the bounded concurrency extension of classical automata given in [DH94], which is based on languages like statecharts, we might wish to increase the cooperation between the different classes by augmenting the transitions in a class with direct (message-less) probe-like queries on the status of other components. This amounts to allowing limited kinds of cross-object broadcasting. For instance, we might have a transition labeled

$$O[j]? \to \sigma/\tau,$$

which would correspond to a transition labeled σ/τ in our object system, but which can be taken only if $O[j]$ is attentive. Similarly, one object could ask about the state of another, the requests it has to fulfill in case it is suspended, etc. Such extensions are possible, yet they may significantly increase the complexity of checking substitutability, as shown in [HKV97]. On the more positive side, orthogonality in statecharts can be used beneficially to model the behavioral aspects that an inheriting class supports beyond those that it inherits. These new pieces of behavior would be modeled naturally by separate orthogonal components added to the original statechart. This idea requires further research, especially into trying to find reasonably moderate syntactic restrictions that would guarantee some measure of substitutability.

Index-dependent behavior In our definition of object systems, the behavior of an instance $O[j]$ is independent of its index j, which is known to it. By adding an indirect reference *self*, which returns the index, we could allow suicidal requests like $(destroy, O, self)$, which actually leads to a deadlock, or more useful ones of the form $(destroy, O, self - 1)$. We could have an object directing requests to its 'next-door neighbor', $O[self + 1]$, which may be very helpful in some applications. Such an extension is possible and relatively easy.

The language of a system In our definition of the language $\mathcal{L}_S^\xi$ of a system, the order of requests in a trace corresponds to the order in which the requests are *fulfilled*. Alternatively, one could define traces of the system to reflect the order in which requests are *made*. Similarly, in our definitions of reactions, the evaluation of indirect references is given when the requests containing them become active. Thus, the evaluation of *min* and *max* is carried out with respect to the existing and attentive components when a request becomes active. Alternatively, one could evaluate indirect references when the request containing them is pushed onto the stack. Such alternative definitions are possible, and they too are quite easy. It is interesting

to contemplate the subtle differences such changes might make in what constitutes behavioral conformity.

5.2 Substitutability

Recall that we capture substitutability as the ability to replace an inherited class by an inheriting class without losing behaviors, which is defined in terms of trace containment or simulation. Our definitions of linear and branching refinement can be easily modified to capture other desired relations between systems. We mention here several such modifications.

Parameterized substitutability Our definition of trace containment requires $f(\mathcal{L}_{S_1}^{\xi}) \subseteq \mathcal{L}_{S_2}^{f(\xi)}$ to hold for every sequence $\xi \in A_1^* \cup A_1^\omega$. One may sometimes want to check the containment only for some special sequences of external requests, say those in some given set $\mathcal{L} \subset A_1^* \cup A_1^\omega$. Such an extension enables to check substitutability with respect to a designated set of events, say when some assumptions are known about the environment. By applying techniques such as intersecting the language of a system with another given language, it is easy to adjust the algorithms for checking refinement relations to handle this extension.

Two-way refinement relations Sometimes, one may want to check a condition stronger than one-way refinement, namely that the behavior of the inheriting and inherited classes are exactly the same. In other words, not only do we not lose behaviors, but we also do not introduce new behaviors. The corresponding refinement relations from concurrency theory are *trace equivalence* and *bisimulation*. Our definitions of linear and branching substitutability can be easily made to be based on these relations, and so can our method for detecting substitutability. As in classical concurrency theory, checking trace equivalence and bisimulation is not harder than checking trace containment and simulation, respectively.

Refinement of requests Consider two classes A and B such that B is inherited from A. Being more refined, the set of simple requests in the guards in B may be richer than that of A. For example, if in the cook example we substitute HUMAN by POLITE-HUMAN, this may involve the introduction of new requests, like *tip*, which could be requested in addition to the existing requests, or *please-prepare*, which could replace the existing *prepare* request. Such a change in the alphabet should be accompanied by a corresponding adjustment in the definition of the refinement relations. We mention here two possible adjustments. Let S_A and S_B be, respectively, the original system with A and the one obtained from it by substituting B for A. One adjustment, which handles the addition of requests, is to project the (larger) alphabet of S_B on requests that are used by both S_A and S_B. Formally, this means that if B can request σ and A cannot, we refer to a $\langle B, \sigma \rangle$-successor of a configuration as a silent successor. The second adjustment, which handles the replacement of requests by more refined ones, is to extend the refinement mapping f to map also requests of B to requests of A. For example, we can map

please-prepare to *prepare*. In the definition of trace containment and simulation, we would then require agreement between requests with respect to f, exactly as we do for the agreements between classes.

Inheritance of control Our definitions of substitutability ultimately ignore control requests and particular instances. One may be interested in inheriting some such elements too, such as the number of instances of a class that have been alive during the execution or the number of instances alive once the execution has terminated. Such tighter relations can be used to check that not only we not lose behaviors, but we also accomplish them in the very same way. They can be tested either by adding orthogonal checks or by extending the alphabets of the systems.

References

[AC95] M. Abadi and L. Cardelli, On Subtyping and Matching, In *Proc. 9th ECOOP*, LNCS 952, pp. 145-167, Springer-Verlag, 1995.

[Boo94] G. Booch. *Object-oriented analysis and design, with applications.* 2nd edition. Benjamin/Cummings, San Mateo, California, 1994.

[CD94] S. Cook and J. Daniels. *Designing object systems: object-oriented modelling with syntropy.* Prentice Hall, New York, 1994.

[DH94] D. Drusinsky and D. Harel. On the power of bounded concurrency I: Finite automata. *Journal of the ACM*, 41(3):517–539, 1994.

[DH99] W. Damm and D. Harel. LSCs: Breathing Life into Message Sequence Charts. *Proc. 3rd IFIP Int. Conf. on Formal Methods for Open Object-based Distributed Systems*, (P. Ciancarini, A. Fantechi and R. Gorrieri, eds.), pp. 293–312, Kluwer Academic Publishers, 1999.

[Har87] D. Harel. Statecharts: A visual formalism for complex systems. *Sci. Computer Prog.*, 8:231–274, July 1987.

[HG97] D. Harel and E. Gery. Executable object modeling with statecharts. *Computer, IEEE Computer Society*, 30(7):31–42, 1997.

[HK99] D. Harel and H. Kugler Synthesizing Object Systems From LCS Specifications. Submitted, 1999.

[HKV97] D. Harel, O. Kupferman, and M.Y. Vardi. On the complexity of verifying concurrent transition systems. In *Proc. 8th Conference on Concurrency Theory*, Lecture Notes in Computer Science, volume 1243, pages 258–272, 1997. Springer-Verlag.

[Jac92] I. Jacobson. *Object-oriented software engineering: a use case driven approach.* Addison-Wesley, Readding, Mass., 1992.

[KV98] O. Kupferman and M.Y. Vardi. Verification of Fair Transition Systems. In *Chicago Journal of Theoretical Computer Science*, 1998(2), March 1998.

[LW94] B.H. Liskov and J.M. Wing, A behavioral notion of subtyping, In *ACM TOPLAS*, 16(1)1811–1841, November 1994.

[LW95] B.H. Liskov and J.M. Wing, Specifications and Their Use in Defining Subtypes LNCS 967, pp. 245–267, 1995.

[Mil71] R. Milner. An algebraic definition of simulation between programs. In *Proc. 2nd International Joint Conference on Artificial Intelligence*, pages 481–489. British Computer Society, September 1971.

[Mil80] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer Verlag, Berlin, 1980.

[Nie95] O. Nierstasz. Regular types for active objects. In *Object-oriented software composition*, pages 99–121. Prentice Hall, New York, 1995.

[Pnu85] A. Pnueli. Linear and branching structures in the semantics and logics of reactive systems. In *Proc. 12th Int. Colloquium on Automata, Languages and Programming*, pages 15–32. Lecture Notes in Computer Science, Springer-Verlag, 1985.

[RJB99] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual.* Addison-Wesley, 1999.

[Rhap] I-Logix, Inc. products web page. http://www.ilogix.com/fs_prod.htm.

[Rum91] J. Rumbaugh et al. *Object-oriented modeling and design.* Prentice Hall, New York, 1991.

[Sou99] J.L. Sourrouille. UML Behavior: Inheritance and Implementation in Current Object-Oriented Languages. *Proc. 2nd Int. Conf. on the Unified Modeling Language*, Oct. 1999.

[Str97] B. Stroustrup. *The C++ Programming Language.* 3rd edition. Addison-Wesley, 1997.

[SVW87] A.P. Sistla, M.Y. Vardi, and P. Wolper. The complementation problem for Büchi automata with applications to temporal logic. *Theoretical Computer Science*, 49:217–237, 1987.

[Uml] Various documents on the Unified Modeling Language (UML), available from the Object Management Group (OMG), http://www.omg.org.

[WZ88] P. Wegner and S.B. Zdonik Inheritance as an incremental modification mechanism or what like is and isn't like. In *Proc. ECOOP*, Lecture Notes in Computer Science, volume 322, pages 55–77, Springer-Verlag, 1985.