Axiomatic Characterization of Trace Reachability for Concurrent Objects

Frank S. de Boer and Hans-Dieter A. Hiep(✉)

Centrum Wiskunde & Informatica, Science Park 123,
1098 XG Amsterdam, The Netherlands
{frb,hdh}@cwi.nl

Abstract. In concurrent object models, objects encapsulate local state, schedule local processes, interact via asynchronous method calls, and methods of different objects are executed concurrently. In this paper, we introduce a compositional trace semantics for concurrent objects and provide an axiomatic characterization of the general properties of reachable traces, that is, traces that can be generated by a concurrent object program. The main result of this paper is a soundness and completeness proof of the axiomatic characterization.

Keywords: Concurrency · Compositionality · Completeness · Program synthesis

1 Introduction

A formal approach for performing verification of large and complex computer systems is compositional when a large system can be decomposed into separate components, each verified independently. Complex systems are constructed by composing verified components together; compositionality ensures that verified properties are preserved in such compositions [4]. Independent verification of components makes the verification work of a whole system divisible, verified components are reusable thus eliminating repeatable work, and therefore this approach is amenable for practical application.

In practice, large-scale computer systems typically consist of many spatially distributed processing units, each with independent and isolated computing and storage capabilities, interconnected by networks. Processing units are dynamically provisioned and interact asynchronously. Such systems are naturally modelled as concurrent object-oriented systems, in which concurrent objects are representing such processing units [7]. Classes of concurrent objects comprise object behaviors that share common behavioral properties, as specified by a concurrent object program. As such a program is executed, configuration and state changes, and a trace of interactions/communications between objects is generated. In general, for concurrent programs, such traces provide the semantic basis for compositional verification and abstraction from internal implementation details [4].

W. Ahrendt and S. L. Tapia Tarifa (Eds.): IFM 2019, LNCS 11918, pp. 157–174, 2019.
https://doi.org/10.1007/978-3-030-34968-4_9

Properties of traces can be expressed in some logical system, say first-order logic. We assume the logic has a satisfaction relation $\theta \models \phi$ which states that the trace θ of communications satisfies the property ϕ. The main contribution of this paper is an axiomatization of the class of properties which hold for any reachable trace describing the interaction between concurrent objects. Thus, properties that hold for every reachable trace can be established axiomatically, abstracting from the programming language semantics.

In this work, specifically, we give a finite axiomatization **Ax** of traces that can be generated by a concurrent object program, and show both *soundness* and *completeness*. The main structure of the completeness proof is as follows.

The semantic consequence relation $\psi \models \phi$ states that any trace θ which satisfies the property ψ, also satisfies property ϕ. That is, $\theta \models \psi$ implies $\theta \models \phi$. Not all traces follow from the axiomatization **Ax**, that is, there are traces θ such that $\theta \not\models \mathbf{Ax}$. Completeness means that a property that holds for all reachable traces, also follows from the axioms. Spelled out this amounts to the following statement: if for every program P and trace θ generated by P we have $\theta \models \phi$, then $\mathbf{Ax} \models \phi$. Assume that for every program P and trace θ of P, we have $\theta \models \phi$. To show that $\mathbf{Ax} \models \phi$, it suffices to show that for *any* trace θ which satisfies all the properties of **Ax** there exists a program P that generates θ (because by the assumption we then have that $\theta \models \phi$). At the heart of the completeness proof therefore lies the synthesis problem which consists of constructing for any trace that satisfies all the properties of **Ax**, a program that generates it.

To illustrate the main ideas underlying the program synthesis, we focus in this paper on an asynchronous active (concurrent) object language [3] that features classes, dynamic object creation, and asynchronous method calls which are immediately stored in a FIFO queue of the callee, later to be dequeued for execution by the callee in a run-to-completion mode. Further, each object encapsulates its local state and executes at most one method at a time.

Plan of the Paper. We first describe the programming language and its semantics in Sect. 2. We introduce the axiomatization and discuss soundness in Sect. 3. Section 4 describes the program synthesis and its use in the completeness proof. In Sect. 5, we sketch out how to extend our result to include futures and cooperative scheduling of method invocations as supported by the Abstract Behavioral Specification language (ABS, see [10]). We will discuss related and future work in Sect. 6.

2 The Programming Language

A *signature* is a finite set of *classes* $\mathcal{C}$, disjoint of a finite set of *methods*, and a map associating methods with classes. A *program* P of some signature consists of a set of class definitions and a designated main class. A *class definition* **class** C $\{\ldots\}$ consists of a class C, a constructor definition $C(\mathbf{arg}) :: s$ and a set of method definitions. A *method definition* $m(\mathbf{arg}) :: s$ consists of a method m and a statement s. Every constructor and method has precisely one argument. Although programs can be supplied with typing information, we do not deal

with type annotations in this paper. We assume that every class C has a unique class definition in program P, and that it contains a unique method definition for every method associated to C. Let α be either a method m or a class C, then we write $P(\alpha :: s)$ to mean there is a definition $\alpha(\mathbf{arg}) :: s$ in program P.

The set of *statements* with typical element s is defined as follows: $s ::=$ $(\mathbf{if}\ e\ \mathbf{then}\ s\ \mathbf{else}\ s) \mid (\mathbf{while}\ e\ s) \mid (s; s) \mid w \coloneqq e \mid w \coloneqq \mathbf{new}\ C(e) \mid e!m(e') \mid \mathbf{skip}$ It consists of standard control structures and assignments, and object creations $(w \coloneqq \mathbf{new}\ C(e))$ and asynchronous method calls $(e!m(e'))$, where w stands for either an instance variable or a local variable, and e for an expression. We treat statements syntactically equivalent up to associativity $(s; (s'; s'')) \equiv ((s; s'); s'')$, and identity $(\mathbf{skip}; s) \equiv s \equiv (s; \mathbf{skip})$, and drop parentheses if not ambiguous.

The variables of a program are drawn from three disjoint sets of *variables*: instance variables with typical element x, local variables with typical element k, and special variables. Variables lack variable declarations in programs, and instance and local variables have an undefined initial value. We further distinguish the special instance variable **this**, which denotes the currently executing object, and the special argument variable **arg**, which denotes the argument to the constructor or method.

By e we denote an arbitrary (side-effect free) *expression*. We have constant terms **null**, **true** and **false**, an equality operator, standard Boolean operators, and w, **this** and **arg** may occur as variable terms.

We present our semantics as a calculus of *finite* sequences of objects and events. The semantics is given with respect to a program P of some fixed signature. There is a countably infinite set O_C of *(object) references* for each class C, such that all O_C are disjoint. We denote by O their union. Thus, all references $o \in O$ have a unique class C such that $o \in O_C$. By V we denote an arbitrary set of *values*, such that $O \subseteq V$ and there are distinct values $\mathbf{null}, \mathbf{true}, \mathbf{false} \in V$ that are not references.

Assignments are partial maps of variables to values. By σ (resp. τ) we denote an assignment of instance (resp. local) variables to values, where **this** (resp. **arg**) is treated as a special variable standing for an object's identity (resp. a method's argument). We say σ is an *object assignment*, and τ is a *local assignment*. We write $[v/\mathbf{this}]$ (resp. $[v/\mathbf{arg}]$) for an *initial* assignment, which assigns **this** (resp. **arg**) to value $v \in V$ and all instance (resp. local) variables are undefined. Furthermore, $\sigma[v/x]$ (resp. $\tau[v/k]$) denotes the *update* of an assignment that assigns instance variable x (resp. local variable k) to value v. So, **this** and **arg** are never updated after their initial assignment.

By $V_{\sigma,\tau}(e)$ we denote the result of evaluating expression e under the assignments σ and τ as usual, and we write $V(e)$ if the subscript is obvious. We assume every non-reference value is *expressible*, that is, for every $v \in V \setminus O$ there is an e such that $V_{\sigma,\tau}(e) = v$ for every assignment σ and τ.

An *object* is either *stable* σ or *active* (σ, τ, s), where σ is an object assignment, τ is a local assignment and s a statement. Stable objects are sometimes also called passive or waiting: they have no active statement and local assignment. For object ξ, by $\xi(\mathbf{this})$ we mean $\sigma(\mathbf{this})$ in either case, and by $(\sigma, \tau, s)[v/w]$ we

mean either $(\sigma[v/x], \tau, s)$ or $(\sigma, \tau[v/k], s)$, in case w is an instance variable x or a local variable k, respectively.

A *(global) configuration* is a sequence of objects separated by $\cdot$ dots. By $\Gamma \cdot \xi$ we mean a global configuration which consists of appending the object ξ to the (possibly empty) global configuration Γ. We treat global configurations syntactically equivalent up to commutativity of its objects.

Events record interactions/communications between objects, and are either:

1. *asynchronous method calls* $o' \rightsquigarrow o.m(v)$,
2. *object creations* $o' \rightsquigarrow o.C(v)$,
3. *root object creations* $\top \rightsquigarrow o.C(v)$,
4. *method selections* $o.m(v)$,
5. *constructor selections* $o.C(v)$,

where o, o' are object references, and v an *argument* value. A *trace* is a sequence of events separated by $\cdot$ dots. Let θ be such a trace, then $|\theta|$ denotes its length.

An asynchronous method call or (root) object creation is an *output* event, and a method or constructor selection is an *input* event. The *(callee) site* of any event is just $o.m(v)$ or $o.C(v)$. We write $o.\alpha(v)$ where α stands for either method m or class C. An output event *corresponds* to an input event if they have the same site. For every site, we assume that method m is associated to the class of o, and that C is the class of o. We leave argument v unrestricted, except for root object creations where we assume $v \notin O$.

We introduce two projections $\theta!o$ and $\theta?o$ which denote the sequence of output events with o as callee, and the sequence of input events with o as callee. The underlying FIFO discipline of processing calls ensures that $\theta?o$ is a prefix of the sites of $\theta!o$. We then define the next site to be executed by an object o, denoted by $Sched(\theta, o)$, as the first site in $\theta!o$ that is not in $\theta?o$, if it exists.

Derivability of $P \Rightarrow (\Gamma, \theta)$ (Definition 1) means that within an execution of program P we can reach configuration Γ by generating the (*global*) trace θ.

The rules operate on two levels: on the global level, the Q-rules activate objects by handling input events, and take active objects back to a stable state. On the local level, the S-rules perform the local computational steps of an active object. The calculus is non-deterministic: multiple choices are allowed by commutativity of global configurations, and picking a fresh reference for objects created by the S_{new} rule.

The rules S_{if}, S_{while} and S_{update} are standard for control structures, modifying the active object in its place. The two S_{if} rules do not apply if $V(e)$ is not **true** and not **false**. Rules S_{asyn} and S_{new} are essentially capturing communication between concurrent objects. Rule S_{asyn} does not apply if $V(e)$ is not a reference, or method m is not associated to the class of $V(e)$. Rule S_{new} creates a new object where $o \in O_C$ is an arbitrary *fresh* reference, that does not occur in the prior configuration or trace. The rule Q_{select} takes a stable object and results in an active object: it applies to both constructor and method selection events, and selects the first pending site after we look up in program P the corresponding method or constructor definition $\alpha :: s$. Rule Q_{skip} takes an active object and results in a stable object: finished continuations are discarded.

The initial configuration consists of a single object $[o/\mathbf{this}]$, and the initial trace consists of a root object creation of the the main class of P. One may think of $\top$ as a dummy reference, but formally $\top \notin O$.

Definition 1. We inductively define $P \Rightarrow (\Gamma, \theta)$ as follows:

$$\frac{P \Rightarrow (\Gamma \cdot (\sigma, \tau, (\mathbf{if}\ e\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2); s)\,,\ \theta) \quad V_{\sigma,\tau}(e) = \mathbf{true}}{P \Rightarrow (\Gamma \cdot (\sigma, \tau, s_1; s)\,,\ \theta)}\ S_{\text{if-true}}$$

$$\frac{P \Rightarrow (\Gamma \cdot (\sigma, \tau, (\mathbf{if}\ e\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2); s)\,,\ \theta) \quad V_{\sigma,\tau}(e) = \mathbf{false}}{P \Rightarrow (\Gamma \cdot (\sigma, \tau, s_2; s)\,,\ \theta)}\ S_{\text{if-false}}$$

$$\frac{P \Rightarrow (\Gamma \cdot (\sigma, \tau, (\mathbf{while}\ e\ \mathbf{do}\ s'); s)\,,\ \theta)}{P \Rightarrow (\Gamma \cdot (\sigma, \tau, \mathbf{if}\ e\ \mathbf{then}\ (s'; \mathbf{while}\ e\ \mathbf{do}\ s'); s\ \mathbf{else}\ s)\,,\ \theta)}\ S_{\text{while}}$$

$$\frac{P \Rightarrow (\Gamma \cdot (\sigma, \tau, w := e; s)\,,\ \theta)}{P \Rightarrow (\Gamma \cdot (\sigma, \tau, s)[V_{\sigma,\tau}(e)/w]\,,\ \theta)}\ S_{\text{update}}$$

$$\frac{P \Rightarrow (\Gamma \cdot (\sigma, \tau, e!m(e'); s)\,,\ \theta) \quad V_{\sigma,\tau}(e) \in O_C}{P \Rightarrow (\Gamma \cdot (\sigma, \tau, s)\,,\ \theta \cdot \sigma(\mathbf{this}) \rightsquigarrow V_{\sigma,\tau}(e).m(V_{\sigma,\tau}(e')))}\ S_{\text{asyn}}$$

$$\frac{P \Rightarrow (\Gamma \cdot (\sigma, \tau, w := \mathbf{new}\ C(e); s)\,,\ \theta) \quad \text{where } o \text{ is fresh}}{P \Rightarrow (\Gamma \cdot (\sigma, \tau, s)[o/w] \cdot [o/\mathbf{this}]\,,\ \theta \cdot \sigma(\mathbf{this}) \rightsquigarrow o.C(V_{\sigma,\tau}(e)))}\ S_{\text{new}}$$

$$\frac{P \Rightarrow (\Gamma \cdot \sigma\,,\ \theta) \quad \mathit{Sched}(\theta, \sigma(\mathbf{this})) = o.\alpha(v) \quad P(\alpha :: s)}{P \Rightarrow (\Gamma \cdot (\sigma, [v/\mathbf{arg}], s)\,,\ \theta \cdot o.\alpha(v))}\ Q_{\text{select}}$$

$$\frac{P \Rightarrow (\Gamma \cdot (\sigma, \tau, \mathbf{skip})\,,\ \theta)}{P \Rightarrow (\Gamma \cdot \sigma\,,\ \theta)}\ Q_{\text{skip}} \qquad \frac{o \in O_C \quad C \text{ is main class}}{P \Rightarrow ([o/\mathbf{this}]\,,\ \top \rightsquigarrow o.C(v))}\ O_{\text{init}}$$

We say $P \Rightarrow (\Gamma, \theta)$ is *derivable* if it can be obtained from above rules.

Our calculus is a "big-step" semantics. To see how our calculus is a trace semantics, we abstract from particular configurations to describe trace reachability:

Definition 2. $T(P)$ denotes the set of reachable traces of program P, that is, $\theta \in T(P)$ iff there is a stable configuration Γ such that $P \Rightarrow (\Gamma, \theta)$ is derivable. A trace θ is *reachable* if there exists a program P such that $\theta \in T(P)$.

A *stable configuration* is a configuration where every object is stable. This technical requirement simplifies the formulation of Axiom 4 in Sect. 3 considerably: otherwise we cannot splice a trace into its segments of input events followed by *all* output events and have to deal with methods that have not yet completed. Moreover, every non-stable configuration can be turned into a stable configuration if every method terminates: let active objects run to completion until a stable configuration is obtained.

For each derivable $P \Rightarrow (\Gamma, \theta)$ and for every reference o, there is at most one object ξ in configuration Γ such that $\xi(\mathbf{this}) = o$. Thus we may treat Γ as a partial map of references to objects, and write $\Gamma(o)$ to denote the unique object ξ for which $\xi(\mathbf{this}) = o$ if it exists. Further, $\Gamma[\xi/o]$ denotes the configuration one obtains from Γ where the object $\Gamma(o)$ is replaced by ξ.

A *local trace* θ_o of a reference o is obtained by the projection of a trace θ to only the events concerning local behavior of o.

$$\begin{aligned}(o \rightsquigarrow o'.\alpha(v) \cdot \theta)_o &= o \rightsquigarrow o'.\alpha(v) \cdot \theta_o \\ (o' \rightsquigarrow o''.\alpha(v) \cdot \theta)_o &= \theta_o \text{ if } o \neq o' \\ (o.\alpha(v) \cdot \theta)_o &= o.\alpha(v) \cdot \theta_o \\ (o'.\alpha(v) \cdot \theta)_o &= \theta_o \text{ if } o \neq o'\end{aligned} \tag{1}$$

This projection records all the outgoing calls generated by o as a caller and all its method and constructor selections. It abstracts from (pending) incoming calls generated by other objects (see Eq. (1) when $o = o''$).

Informally, objects are globally indistinguishable when generating the same local trace (see [2] for a formal treatment). For this, we consider two (local or global) traces *equivalent modulo renaming* when there exists a renaming that makes one of the traces equal to the other. A *renaming* λ is a family of bijections $\{f : O_C \to O_C\}_{C \in \mathcal{C}}$. By applying a renaming to configurations and traces, we substitute every occurrence of a reference $o \in O_C$ by $f(o)$ for the corresponding $f : O_C \to O_C$ in λ. For example, for distinct $o, o', o'' \in O$, the global traces

$$\begin{aligned}&\top \rightsquigarrow o.C(\mathbf{true}) \cdot o.C(\mathbf{true}) \cdot o \rightsquigarrow o'.C(\mathbf{false}) \cdot o \rightsquigarrow o'.m(o') \cdot o'.C(\mathbf{false}) \\ &\top \rightsquigarrow o'.C(\mathbf{true}) \cdot o'.C(\mathbf{true}) \cdot o' \rightsquigarrow o.C(\mathbf{false}) \cdot o' \rightsquigarrow o.m(o) \cdot o.C(\mathbf{false})\end{aligned}$$

are equivalent modulo renaming. The dummy reference is never renamed. Since we have that the set O of object references is countable, there exists a natural way of mapping references to natural numbers. Consider an encoding of traces, à la N.G. de Bruijn, that incrementally replaces every distinct reference by an index. For example, both traces above have the following encoding:

$$\top \rightsquigarrow \mathbf{1}.C(\mathbf{true}) \cdot \mathbf{1}.C(\mathbf{true}) \cdot \mathbf{1} \rightsquigarrow \mathbf{2}.C(\mathbf{false}) \cdot \mathbf{1} \rightsquigarrow \mathbf{2}.m(\mathbf{2}) \cdot \mathbf{2}.C(\mathbf{false})$$

In different local traces, the indices that encode the same reference may differ depending on the position where it occurs first in a local trace. The same index may be used to encode different references in different traces. In non-empty local traces of a reachable trace, indices of created object are never $\mathbf{1}$, and $\mathbf{1}$ always encodes **this**.

3 Axiomatization

The following general properties describe the FIFO ordering between corresponding events, uniqueness of object creation, consistent data flow, and determinism.

The following axiom states that method calls are selected for execution in a FIFO manner: every input event corresponds to a unique prior output event.

Axiom 1 (FIFO)**.** Let $\mathbf{F}$ denote the property such that $\theta \models \mathbf{F}$, if for every object reference o and every prefix θ' of θ, it holds that $\theta'?o$ is a prefix of $\theta'!o$.

An object reference o is created by an event $o' \rightsquigarrow o.C(v)$ (or by $\top \rightsquigarrow o.C(v)$, in case of the root object), and we then say that o' created o. We have the following axiom which guarantees uniqueness of object references.

Axiom 2 (Creation)**.** Let $\mathbf{C}$ denote the property such that $\theta \models \mathbf{C}$, if: (1) the first and only the first event of θ is a root object creation, (2) all references o occurring in θ have been created exactly once, (3) an object cannot create other objects before it has been created itself, and (4) an object cannot create itself.

It is worthwhile to note that the FIFO axiom alone does not rule out that an object can create other objects before it has been created itself. Moreover, we will argue later that the FIFO and Creation axioms together with the following axiom rule out any calls by an object before it has been created.

Let $K_o(\theta)$ denote all the object references that o has created in θ or occur in $\theta?o$. The following axiom captures that every outgoing call of an object contains only prior information about objects that o has created, or that have been acquired by the parameter of a method selection.

Axiom 3 (Knowledge)**.** Let $\mathbf{K}$ denote the property such that $\theta \models \mathbf{K}$, if for every trace θ' and every reference o, o', o'' and value $v \notin O$ the following holds: (1) if $\theta' \cdot o \rightsquigarrow o'.m(v)$ is a prefix of θ then o' is in $K_o(\theta')$, (2) if $\theta' \cdot o \rightsquigarrow o'.m(o'')$ is a prefix of θ then both o' and o'' are in $K_o(\theta')$, and (3) if $\theta' \cdot o \rightsquigarrow o'.C(o'')$ is a prefix of θ then o'' is in $K_o(\theta')$.

The final axiom describes the local determinism of objects, namely, that output behavior of objects is completely determined by its inputs. To formulate this axiom, consider that every local trace θ_o can be sliced into segments, such that each *segment* starts with an input event followed by as many output events as possible: all segments put back in order forms the local trace we started with. The number of input events of $\theta?o$ is equal to the number of segments of θ_o.

Axiom 4 (Determinism)**.** Let $\mathbf{D}$ denote the property such that $\theta \models \mathbf{D}$, if for every object reference o', o'' and every trace θ', θ'' such that θ' is a prefix of $\theta?o'$ and θ'' is a prefix of $\theta?o''$ and θ' is equivalent modulo renaming to θ'', the first $|\theta'|$ segments of $\theta_{o'}$ and $\theta_{o''}$ are equivalent modulo renaming.

We illustrate the use of the above axioms by the following properties.

Proposition 1. *For every reference o occurring in θ, there is no output event with o as caller or input event with o as callee* before *the creation event of o.*

Proof. Suppose that o calls a method on another object o', before it has been created. By the Knowledge axiom, o' either must have been created by o or received as parameter before the outgoing method call by o. That o creates another object o' before it has been created itself is ruled out by the Creation axiom. So o must have received o' as a parameter before the outgoing call, and

thus before it has been created. But the selection of a method by o *before* the creation event of o leads to an infinite regression: first observe that the FIFO axiom implies the existence of a corresponding (preceding) method call. The Creation and Knowledge axioms in turn imply that the caller of that method call to o has received o as a parameter of a previous method selection. This leads to an infinite sequence of method calls which is obviously impossible as each trace has only a finite number of events. □

Proposition 2. *For every reference appearing in θ, only the first event of its local trace is a constructor selection.*

Proof. If θ_o is empty, we are done. So suppose θ_o is non-empty, and consider its first event. It cannot be an asynchronous call with o as caller, nor the selection of an incoming call to o before its creation event, as these cases are ruled out by Proposition 1. When there is a pending call to o that call must be performed after the creation event of o, and by the FIFO axiom, the first event is the constructor selection. □

Proposition 3. *For every prefix of θ that ends with an object creation of o', there is no occurrence of o' before that creation event.*

Proof. By contradiction: suppose o' occurs in an event before $o \rightsquigarrow o'.C(v)$. It cannot occur as the callee of a creation event by Creation axiom, it cannot occur as an argument of an output event or as the callee of a method call as o' cannot appear in the learned knowledge of any other object before its creation event, and it cannot occur as the caller of an output event or as callee of an input event by Propositions 1 and 2 and the FIFO axiom. These are all possible cases. □

By axiomatization **Ax** we mean the above axioms: **F**, **C**, **K**, **D**. Let ϕ be a property, i.e. a predicate on *arbitrary* traces. A property ϕ is *R-valid* if $\theta \models \phi$ for every *reachable* trace θ (see Definition 2). We have the main soundness theorem.

Theorem 4 (Soundness)**.** *All the axioms of* **Ax** *are R-valid.*

Assuming an arbitrary program it suffices to show that all its reachable traces satisfy the axioms of **Ax**, which can be established by a straightforward though tedious induction on the length of the derivation. As a corollary we infer that $\mathbf{Ax} \models \phi$ implies that ϕ is R-valid. In other words, every property that follows from the axioms holds for every reachable trace.

4 Program Synthesis

The synthesis problem is to construct a program from a trace which satisfies the axioms **Ax**, such that the program witnesses that the given trace is *reachable*. We assume well-typed traces so that the class and method signature of the synthesized program coincides with the given trace. From the trace we then need to synthesize the method bodies (including the bodies of the constructors) which are expected to provide a general template for all the individual object behaviors as they appear in the trace.

We observe that each local trace, that is the individual object behavior, can be sliced into a number of segments, which consist of an input event followed by a largest possible sequence of consecutive output events. Since every method runs to completion, the output of a segment following an input event must be generated by the method selected by that input event. So we must define a method body in some way as a *choice* between *all* it corresponding segments. This choice requires knowledge of the executing object and the position of each segment in its local trace, which we will represent by an instance variable that serves as a program counter.

A large part of difficulty in solving the synthesis problem comes from the fact that traces contain object references, while programs never directly access such references. However, our programming language does have an equality operator suitable for checking whether two object references are equal. An object may thus learn of the existence of a new object reference, acquired by the argument of a method selection, by comparing it with any other reference it has encountered before. Moreover, values that are not references can be synthesized directly, by the assumption that such values are expressible.

The notion of knowledge is local to an object, since the knowledge of one object is not directly accessible to another object. When an object receives an unknown reference, it cannot determine its exact identity. Thus, under similar circumstances, two objects of the same class that both receive two unknown references that are actually different must be treated as if they were identical, since there is no expression that can tell the actual identity of an object.

An object may perform an outgoing call with as argument a reference that it learned by the argument of any of its previous method selections or by the creation of a new object. We need to ensure that the state of an object, the assignment σ of instance variables to values, reflects the current knowledge of an object. To do so, we update instance variables whenever a (previously unknown) reference is encountered: this happens possibly after a method selection and definitely after object creation. However, how do we determine in what instance variables such learned references are stored?

In order to abstract from the actual object references as they appear in a given trace, the program is synthesized using the above De Bruijn-like encoding[1] for the definition of its (object) instance variables. The encoding gives to each object reference a unique index: obtained by the first occurrence of that reference, counting from left-to-right. An encoded local trace represents an equivalence class of the local behavior of objects. Although two objects can have different behavior, their local behavior may differ only after a number of input events. For the time where the two objects have received equivalent input, the behavior of the two objects must be the same too, since they run the same method bodies to completion. For that reason, our encoding coincides on two equivalent prefixes of local traces.

[1] We say De Bruijn-*like* to mean the same purpose that De Bruijn had in Automath [5]: namely to identify two renaming-equivalent terms. The technical nature of De Bruijn-indices, e.g. in lambda calculus, is different than from here.

Each class C uses the instance variable x_{pc} for the program counter which value corresponds to the current segment. It further uses $x_1, \ldots, x_n$ instance variables, where n is the maximum number of object references which appear in a local trace of an instance of C. The variable x_i represents the storage of an object reference that was encoded by the index i.

The above leads to the following program synthesis. Let θ be a given trace that satisfies the axioms, i.e. $\theta \models \mathbf{Ax}$. By $P(\theta)$ we then denote the resulting synthesized program, that is constructed as follows:

1. For each class C, we collect all prefixes of its local traces in the set $T(\theta, C)$. Formally, $\theta' \in T(\theta, C)$ iff θ' is a prefix of θ_o for some $o \in O_C$.
2. Apply the De Bruijn-like encoding to every trace in $T(\theta, C)$.
3. Construct a forest from the set $T(\theta, C)$ by the prefix order on encoded traces.
4. In the forest, give a unique label to each input event.
5. We then synthesize the constructor and method bodies:
   (a) Do a case distinction on the current position and argument.
   (b) Set the new current position, based on testing the argument.
   (c) Store newly learned knowledge from argument in the state.
   (d) Realize output events of the segment corresponding to current position.
   (e) Store new knowledge from created objects in the state.
6. Declare the class of the root object creation event the main class.

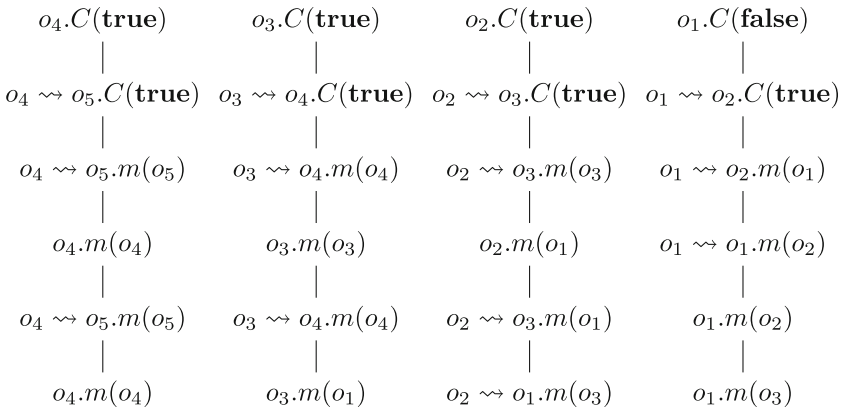

Fig. 1. Example local traces: $\theta_{o_4}, \theta_{o_3}, \theta_{o_2}, \theta_{o_1}$.

We explain the steps of our construction in more detail below, alongside an example trace θ. In our example we deal with only one class: in general, for each class a similar treatment is given. The trace θ starts with the root object creation $\top \rightsquigarrow o_1.C(\mathbf{false})$, and has non-empty local traces $\theta_{o_1}, \theta_{o_2}, \theta_{o_3}, \theta_{o_4}$, and the other local trace, θ_{o_5}, is empty. Only the local traces are shown in Fig. 1. This is also the result of collecting the local traces per class (Step 1).

(Step 2) We bring the local traces in a particular form: we apply an encoding to ensure that two local traces that are equivalent modulo renaming are identified. By the way we encode each object reference with an increasing index, it is ensured that two local traces that have a common prefix modulo renaming have the same encoding on that prefix. Figure 2 shows the encoded traces of the local traces of Fig. 1. These encodings have different interpretations per local trace: e.g. in θ_{o_1} we have $\mathbf{1} \mapsto o_1$, $\mathbf{2} \mapsto o_2$, $\mathbf{3} \mapsto o_3$, but in θ_{o_2} we have a different one: $\mathbf{1} \mapsto o_2$, $\mathbf{2} \mapsto o_3$, $\mathbf{3} \mapsto o_1$.

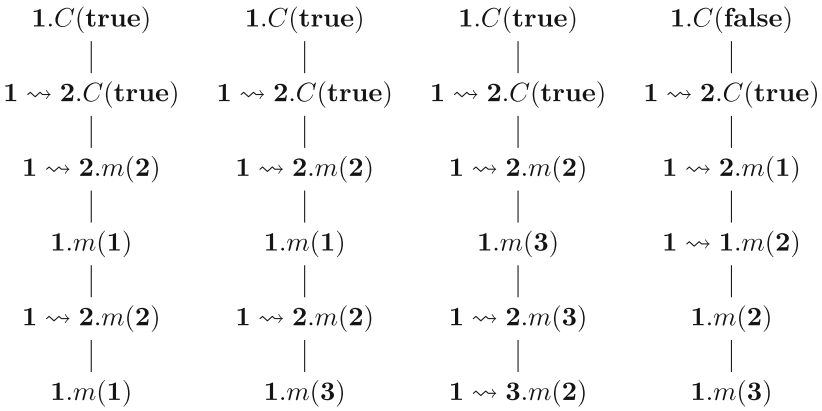

Fig. 2. Result of the encoding of the local traces of Fig. 1.

(Step 3) The set of local traces $T(\theta, C)$ allows us to construct a forest: nodes correspond to events, edges describe the order between events within a local trace. Roots correspond to a constructor selection event, and every path from a root to a leaf corresponds to a local trace. Every two traces that have a common prefix also have a common ancestor in the forest: the last event of the common prefix. It is crucial to construct the forest only *after* encoding the local traces. Figure 3 shows the forest upside down.

(Step 4) We find a labeling of every input event in the forest, by numbering input events in a breath-first left-to-right manner (as shown in Fig. 3 by the circled numbers). Observe that each segment of a local trace corresponds to a path segment in the forest, since it comprises the same events. This correspondence becomes important later, when we realize the output events. Our labeling is used as a *program counter*, to keep track of the possible local traces that an object simulates: at any prefix the object may not yet know which path it will take in the future. In other words, if two different paths from root to leaf have a common ancestor, the object that simulates either of them does not yet know which object it simulates until a choice is made after the common ancestor.

(Step 5) For a given class, we now synthesize its constructor and method bodies. We first describe snippets of statements that implement a particular feature of the construction, and then give the overall construction of class definitions

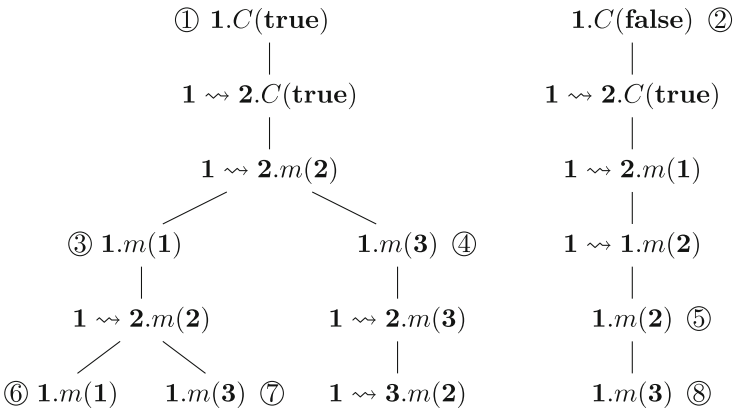

Fig. 3. Forest with shared common prefixes and numbered input events.

as a scheme that pastes these snippets together. As already described above, each class uses the following instance variables: x_1 up to x_n corresponding to encoded references, and x_{pc} for the program counter which value is the current path segment.

Testing the state and argument and set the current position in the forest: after an input event, we check the program counter and argument to decide at which place in the forest the object currently is. After a constructor selection, the program counter has an undefined initial value, and we only inspect the argument. For our example we have these tests:

$test_C =$ **if** $\mathbf{arg} = \mathbf{true}$ **then** $x_{\text{pc}} := 1$
else if $\mathbf{arg} = \mathbf{false}$ **then** $x_{\text{pc}} := 2$
else skip

$test_m =$ **if** $x_{\text{pc}} = 1 \wedge \mathbf{arg} = \mathbf{this}$ **then** $x_{\text{pc}} := 3$
else if $x_{\text{pc}} = 1$ **then** $x_{\text{pc}} := 4$
else if $x_{\text{pc}} = 2 \wedge \mathbf{arg} = x_2$ **then** $x_{\text{pc}} := 5$
else if $x_{\text{pc}} = 3 \wedge \mathbf{arg} = \mathbf{this}$ **then** $x_{\text{pc}} := 6$
else if $x_{\text{pc}} = 3$ **then** $x_{\text{pc}} := 7$
else if $x_{\text{pc}} = 5$ **then** $x_{\text{pc}} := 8$
else skip

There are three kinds of tests on arguments: a value tested for equality (e.g. $\mathbf{arg} = \mathbf{true}$), a known reference is checked (e.g. $\mathbf{arg} = \mathbf{this}$ or $\mathbf{arg} = x_2$), or a reference is unknown (e.g. when $x_{\text{pc}} = 1$ or 3 or 5). The last kind of test is checked after all known references are checked, since if all tests of known references fail the reference must be unknown. For method selections, we check the previous value of the program counter to decide which conditions to check

(e.g. for $x_{\mathrm{pc}} = 1$ only the input events 3 and 4 could match). We never test for possibilities that do not occur in our forest.

Case distinction on the current position: after the former step of deciding the current position, we will perform three other tasks: if applicable, store the newly learned knowledge from the argument in an instance variable. Then we perform the output events by asynchronous method calling and object creation statements, and further store the newly learned knowledge from object creations in instance variables. The purpose of performing a case distinction on the current position is to realize output events that correspond to a segment. For example:

$out_C =$ **if** $x_{\mathrm{pc}} = 1$ **then**
  $x_2 \coloneqq$ **new** C(**true**);
  $x_2!m(x_2)$
else if $x_{\mathrm{pc}} = 2$ **then**
  $x_2 \coloneqq$ **new** C(**true**);
  $x_2!m$(**this**);
  this$!m(x_2)$
else skip

$out_m =$ **if** $x_{\mathrm{pc}} = 3$ **then**
  $x_2!m(x_2)$
else if $x_{\mathrm{pc}} = 4$ **then**
  $x_3 \coloneqq$ **arg**;
  $x_2!m(x_3)$;
  $x_3!m(x_2)$
else if $x_{\mathrm{pc}} = 5$ **then**
  skip
else if $x_{\mathrm{pc}} = 6$ **then**
  skip
else if $x_{\mathrm{pc}} = 7$ **then**
  $x_3 \coloneqq$ **arg**
else if $x_{\mathrm{pc}} = 8$ **then**
  $x_3 \coloneqq$ **arg**
else skip

For object creation events we synthesize a **new**-statement and store the reference in the instance variable corresponding to the encoded index as it occurs the local trace. For asynchronous method calls, we simply perform that action. For both output events, argument values are synthesized by constants, and argument references (and the callee reference) are looked up by its corresponding instance variable (or the special **this**-variable for the first occurrence **1**, that always refers to the reference of the object itself). Note how segments $4, 7, 8$ first store the argument, before realizing any output (if any).

The synthesized program has a unique class definition for every class C: we take $C(\mathbf{arg}) :: test_C; out_C$ as constructor definition and $m(\mathbf{arg}) :: test_m; out_m$ as method definition for each method m. Finally, the class that appears in the root object creation of our trace will be taken as main class of the program.

Completeness. Correctness of the above construction requires the following auxiliary notions which allow us to construct explicitly the global configuration that is used as witness for reachability. For every object reference o that occurs in a global trace θ, we know that the local trace θ_o can be encoded in a De Bruijn-like way as described above. Formally, by $e(\theta, o) : O \to I$ we denote the partial function that encodes the occurring references in θ_o as an index, and by

$d(\theta, o) : I \to O$ we denote the partial function that decodes the indices to the actual object reference, such that $d(e(o')) = o'$ for every reference o' that occurs in θ_o, where we write $e(\theta, o)$ and $d(\theta, o)$ simply as e and d since θ and o are clear from context.

Also for every object reference o of class C that occurs in θ, there is a path θ_o from root to leave in the forest $T(\theta, C)$. As explained above, the forest $T(\theta, C)$ associates to each input event node a unique label: by $\ell(\theta' \cdot o.\alpha(v))$ we denote that label for each prefix $\theta' \cdot o.\alpha(v)$ of θ_o. The construction of this forest has the property that there are only choice branches *before* input events: this property follows from Determinism Axiom 4.

The function f will construct the local state of object o, which will be contained in a global configuration corresponding to the trace θ. Recall that for each index $i \in I$ there is an instance variable x_i, and x_{pc} is the program counter. We treat the first occurrence in a local trace specially, as it can always be retrieved by **this**. In the following, let ϵ be the empty trace:

$$
\begin{aligned}
f(\theta_o \cdot o \rightsquigarrow o'.D(v)) &= f(\theta_o)[o'/x_{e(o')}] \\
f(\theta_o \cdot o \rightsquigarrow o'.m(v)) &= f(\theta_o) \\
f(\theta_o \cdot o.\alpha(v)) &= \begin{cases} f(\theta_o)[\ell(\theta_o \cdot o.\alpha(v))/x_{\mathrm{pc}}][v/x_{e(v)}] & \text{if } v \in O \\ f(\theta_o)[\ell(\theta_o \cdot o.\alpha(v))/x_{\mathrm{pc}}] & \text{otherwise} \end{cases} \\
f(\epsilon) &= [o/\mathbf{this}]
\end{aligned}
$$

The above function results in stable objects. Furthermore, f is also well-defined for all prefixes of local traces. We have that for any *non-empty* prefix θ' of θ_o, the set of references contained in object $f(\theta')$ equals the set $K_o(\theta')$. Note that for objects that are created but have not yet selected the constructor, their knowledge set is empty but their state is already $[o/\mathbf{this}]$.

Lemma 5. *Every trace θ for which $\theta \models \mathbf{Ax}$ holds, is reachable.*

Proof. We assume $\theta \models \mathbf{Ax}$ holds. To show that θ is reachable, we show there exists a stable configuration Γ such that $P(\theta) \Rightarrow (\Gamma, \theta)$, where $P(\theta)$ denotes the synthesized program, as described above. We construct a particular global configuration out of objects which are obtained by the above function of local traces, and use that as witness for Γ. In particular, that configuration consists of a sequence of objects $f(\theta_o)$ for every reference o that occurs in θ.

What remains is to show that $P(\theta) \Rightarrow (\Gamma, \theta)$ is derivable (see Definition 1). We prove the existence of such a derivation by induction of the length of θ. By the Creation Axiom 2, the base case is captured by an application of the rule O_{init} that generates the root object creation and the initial configuration.

For any proper prefix θ' of θ, the induction hypothesis states the existence of a derivation $P(\theta) \Rightarrow (\Gamma', \theta')$, such that for each object reference o in θ' there is an object $\Gamma'(o)$. In either case that $\Gamma'(o) = \sigma$ or $\Gamma'(o) = (\sigma, \tau, s)$, we have that $\sigma = f(\theta'_o)$, meaning that the object is in a well-known state as constructed from its local trace. If the local trace is empty, the object has not yet selected

its constructor. Otherwise, we know that its state has a valid program counter: this points to a particular input node in the forest, corresponding to the last selected input event. For active objects, we can relate the current statement even further to an output node in the forest, since the synthesized program is constructed along the same forest. For stable objects, there is no next output node and the object waits for the next method selection. Note that although our goal configuration Γ is stable, the intermediary Γ' is not necessarily stable. The induction step consists of a case analysis of the different next events.

1. The next event is either a constructor selection $o.C(v)$ or method selection $o.m(v)$. By FIFO Axiom 1, we know the construction selection corresponds to a prior creation event that is pending. There exists a stable object configuration $\Gamma(o)$ on which we apply the rule Q_{select} to generate the selection event. In case of a method selection, we know that the current program counter value is defined: the synthesized program first checks which branches in the forest have to be checked (as described in detail above). After running the synthesized body until its first output event or until completion, we have assigned the program counter and updated the knowledge acquired by the argument if v was a reference and not encountered before.
2. The next event is $o \rightsquigarrow o'.m(v)$. By Proposition 1, we know that o must have been created before this output event. Moreover, the object $\Gamma(o)$ is an active object, where the next statement is an asynchronous method call by construction: we apply the rule S_{asyn}, where we evaluate the callee expression and argument expression. By Knowledge Axiom 3, we know that o' is in $K_o(\theta')$ and since the local history of θ_o is non-empty, this equals $f(\theta'_o)$. Hence, the callee expression $x_e(o')$ results in the object reference o'. Similar for argument references.
3. The next event is $o \rightsquigarrow o'.C(v)$. The object $\Gamma(o)$ is an active object, where the next statement is an object creation statement by construction: the resulting reference will be stored in the corresponding instance variable $x_e(o')$. We know that o' is fresh in the current trace by the FIFO axiom, Creation axiom and Knowledge axiom. Thus we can apply S_{new}, where we choose o' as fresh reference, to add $[o'/\mathbf{this}]$ to the global configuration (being $f(\epsilon)$ for o'). For the argument, similar as above. □

Finally, we can state our completeness result with a straightforward proof.

Theorem 6 (Completeness)**.** *If ϕ is R-valid then* $\mathbf{Ax} \models \phi$.

Proof. Let ϕ be R-valid. We have to show that $\mathbf{Ax} \models \phi$. Assume $\theta \models \mathbf{Ax}$. Now, by Lemma 5, trace θ is reachable. Since ϕ is R-valid, we conclude that $\theta \models \phi$. □

5 Future Extensions

In this section we sketch how to extend our axiomatization to cooperative scheduling and futures. First, we consider cooperative scheduling by means of

Boolean **await**-statements. Semantically such statements require the inclusion of a set Q of suspended processes (τ, s): an object ξ is either stable (Q, σ) or active (Q, σ, τ, s). We have the following semantic rules rules Q_{await} and Q_{resume} for the suspension and resumption of processes:

$$\frac{P \Rightarrow (\Gamma \cdot (Q, \sigma, \tau, \textbf{await } e; s), \theta)}{P \Rightarrow (\Gamma \cdot (Q \cup \{(\tau, \textbf{await } e; s)\}, \sigma, \theta)} \; Q_{\text{await}}$$

$$\frac{P \Rightarrow (\Gamma \cdot (Q \cup \{(\tau, \textbf{await } e; s)\}, \sigma), \theta) \quad V_{\sigma,\tau}(e) = \textbf{true}}{P \Rightarrow (\Gamma \cdot (Q, \sigma, \tau, s), \theta)} \; Q_{\text{resume}}$$

Clearly the extended language gives rise to non-determinism in the sense that the input events no longer determine the local behavior of an object because of the non-deterministic internal scheduling of the methods. We must therefore weaken Axiom 4 to apply only for the first segment. Note that the first segment corresponds to outputs which can be generated by the constructor method only and these outputs only depend on the input parameter. The main construction underlying the above program synthesis crucially exploits the general determinism. Therefore, instead of synthesizing the individual method bodies in terms of the output events of the given trace, all the output behavior of all the instances of a given class is synthesized in the body of the constructor. The body of the constructor is in this setting a loop, in which the statement $b :=$ **true**; **await true**; $b :=$ **false** precedes a large case distinction on the program counter, and each case consists of a sequence of method calls (corresponding to the sequence of output events of a segment). The Boolean instance variable b controls the internal scheduling, as described below. When the constructor releases control, a new method can be selected (as expected by the given trace) which first checks and updates the local state as before. After that, the method enters a loop the body of which consists of an update of the program counter followed by an **await true** statement, which may release control. The loop condition is simply $b =$ **true**. The updates of the program counter systematically generates all its possible values (as generated from the given trace). As soon as the constructor method is scheduled again, it thus can proceed with a non-deterministically generated value of the program counter (which corresponds to the number of iterations of the above loop). Note that when the method is scheduled again the above loop will terminate. In showing reachability, one can steer towards the right number of times a method is (re)scheduled to set the counter to the right value.

Next, we briefly sketch the extension to futures, as employed by the ABS language (see also [1]), where futures are dynamically generated references to return values and as such provide an asynchronous communication of these values. This extension consists of a further refinement of the above scheme. It introduces two kinds of events: the completion of a future (by a return statement) and a successful **get** operation on a future. Both input and output events record as additional parameter the future which uniquely identifies the event. Further, we have additional axioms which state that a completion of a future has to be preceded by the corresponding method selection, and that a get event

is preceded by the corresponding completion event. The Creation axiom has also to be extended to ensure the uniqueness of futures, so that every future recorded in an asynchronous method call is unique.

In solving the synthesis problem, futures can only be completed by the return statement of the corresponding method. But as output behavior is realized within the constructor, the constructor method has to release control back to the corresponding method. Each method body is augmented with an "exit protocol" which precedes the return statement and which consists of the statement **await** $r = d$, where r is an instance variable which is set by the constructor and which stores the future that should be completed. The local variable d, the so-called destiny variable, is an implicit formal parameter of every method which hold its own future, that is, the unique future for returning a value.

A more formal treatment of these extensions is left for future work.

6 Discussion

To the best of our knowledge this is a first sound and complete axiomatization of trace reachability for concurrent objects. Doveland et al. [7] present a proof theory for concurrent objects that uses traces for compositional reasoning. Their traces satisfy certain well-formedness conditions, but completeness of their well-formedness conditions is not mentioned. Din and Owe [6] present a soundness and completeness of a proof system for reasoning about asynchronous communication with shared futures. Their proof system also is based on well-formed traces, but soundness and completeness focuses on program correctness instead.

It is worthwhile to note the analogy between our completeness proof and existing completeness proofs of a variety of logics (first-order logic, modal and intuitionistic logics) which in general are based on the so-called "model existence theorem" (see [9]). In our completeness proof the program synthesis corresponds to such a theorem which states the existence of a model for a theory which satisfies certain consistency requirements.

We are currently working on mechanically verifying the established results, formalizing the presented results in the context of the Lean interactive theorem prover. The bulk of this on-going work comprises proving soundness of the axioms, and proving our solution to the synthesis problem correct.

Further ahead is a trace logic that allows the formulation of user-defined class specifications and global invariants. This requires formalization of the trace logic itself, resulting in a trace specification language. The axioms as given above can be formulated in the trace logic, and a corresponding proof system can then establish general properties of reachable traces. Such a system can be integrated with proof methods for verifying the correctness of concurrent object programs, where class definitions are annotated with invariants on the properties of local traces of its objects, and a user-defined global invariant that describes properties of global traces. Any property that follows from the axioms can thus be inferred directly, without requiring such properties to be encoded in global invariants.

Our program synthesis is a step towards the full abstraction problem, that yet remains to be addressed for active object languages. Another application

of the program synthesis is to provide a formal model for generating testing environments. A test case describes the desirable properties of traces, similar to Ducasse et al. [8] By synthesizing a program that reproduces a trace that satisfies the desirable property, and substituting in the resulting program a class under test, allows one to test a class definition in isolation of the rest of a system: we can test whether the desired trace is indeed reachable after such a substitution. This could lead to practical applications by the integration of above methods.

Acknowledgements. Thanks to Lars Tveito for visiting our research group and for having many interesting discussions, Roy Overbeek for reading a draft, and the anonymous referees for their suggestions.

References

1. de Boer, F.S., Clarke, D., Johnsen, E.B.: A Complete guide to the future. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 316–330. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71316-6_22
2. de Boer, F.S., de Gouw, S.: Compositional semantics for concurrent object groups in ABS. Principled Software Development, pp. 87–98. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-98047-8_6
3. de Boer, F.S., et al.: A survey of active object languages. ACM Comput. Surv. **50**(5), 76:1–76:39 (2017)
4. de Roever, W.P., et al.: Concurrency Verification: Introduction to Compositional and Noncompositional Methods. Cambridge Tracts in Theoretical Computer Science, vol. 54. Cambridge University Press, Cambridge (2001)
5. Dechesne, F., Nederpelt, R.: N.G. de Bruijn (1918–2012) and his road to automath the earliest proof checker. Math. Intell. **34**(4), 4–11 (2012)
6. Din, C.C., Owe, O.: A sound and complete reasoning system for asynchronous communication with shared futures. J. Logical Algebraic Methods Program. **83**(5), 360–383 (2014)
7. Dovland, J., Johnsen, E.B., Owe, O.: Verification of concurrent objects with asynchronous method calls. In: IEEE International Conference on Software - Science, Technology and Engineering (SwSTE 2005), 22–23 February 2005, Herzelia, Israel, pp. 141–150 (2005)
8. Ducasse, S., Girba, T., Wuyts, R.: Object-oriented legacy system trace-based logic testing. In: Conference on Software Maintenance and Reengineering (CSMR 2006), pp. 37–46. IEEE (2006)
9. Fitting, M.: Proof methods for modal and intuitionistic logics. J. Symbolic Logic **50**(3), 855–856 (1985)
10. Johnsen, E.B., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M.: ABS: a core language for abstract behavioral specification. In: Aichernig, B.K., de Boer, F.S., Bonsangue, M.M. (eds.) FMCO 2010. LNCS, vol. 6957, pp. 142–164. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-25271-6_8