Verifying OCL specifications of UML models : tool support and compositionality

Kyas, M.

Chapter 6

A Compositional Trace Logic for Behavioural Interface Specifications

We describe a compositional trace logic for behavioural interface specifications and corresponding proof rules for compositional reasoning. The trace logic is defined in terms of axioms in higher-order logic. This trace logic is applicable to any object-oriented programming language. We treat object creation without observing the explicit act of creation.

We prove a soundness result of this approach using the theory of Galois connections. We show the correctness of a specification of the Sieve of Eratosthenes using the proposed method. This notion of compositionality allows the verification of systems during the early stages of a design.

6.1 Introduction

We present a theory for reasoning compositionally about behavioural interfaces for class-based object-oriented programs. Our main contribution is an axiomatic characterisation of unbounded object creation in terms of communication traces over the visible operations of a class (its *signature*). This involves an abstraction of the actual creation of objects as represented explicitly in, for example, Ábrahám [3] for multi-threaded Java.

We apply this method to verifying the correctness of the Sieve of Eratosthenes prime number generator using PVS (cf. Owre [121]). This example involves unbounded object creation.

The next section characterises our notion of an *interface* and *interface invariants*. Section 6.3 describes the *trace logic* used to specify interface invariants. Section 6.4 explains how to reason compositionally about object-oriented systems using interface invariants. Section 6.5 describes the axioms of our trace logic and contains a correctness result based on Galois connections. Section 6.6 applies our method to the "Sieve of Eratosthenes" example. Section 6.7 relates our work to previously published work. Finally, in Section 6.8 we draw our conclusion on our method and discuss future work.

6.2 Interfaces

An *interface* defines the interaction between software components by means of properties of other software components, which abstract and encapsulate their data. This includes constants, data types, and the signature of synchronous and asynchronous message exchange, including exceptions specifications. The interface of a component is deliberately kept separate from its implementation. Any other software component B, which can be referred to as the client of A, that interacts with A is forced to do so only through the interface. The advantage of this arrangement is that any other implementation of the interface of A should not cause B to fail, provided that its use of A complies with the specification of A's interface. Synchronous operations are executed by means of the standard rendez-vous mechanism, resulting in synchronous message exchange; asynchronous operation calls are executed by storing a corresponding message in the receiver's message queue.

Furthermore, an interface specifies a *protocol* between objects, that is, how unrelated objects communicate with each other. In our case a protocol is a description of:

- The messages understood by an object.
- The arguments that these messages may be supplied with.
- The results that these messages return, if any.
- The *invariants* that are preserved despite of the modifications to an object's state.
- The exceptional situations that will be required to be handled by clients of the object.
- The allowed sequence of messages.
- Restrictions placed on either participant in the communication.
- Expected effects that will occur if a message is handled.

In this paper, interfaces are specified using *interface invariants*. These are invariants on *traces of events*, that is, sequences of occurrences describing message sending and receiving, using the trace logic of the following section. Note that whereas we require that interface specifications are compatible with the class of each object, we do not discuss type-checking of interface specifications. Therefore, we do not define the signature of a message, an operation call, or a class.

6.3 Trace Logic

Trace logic is used to specify and verify properties of the externally observable behaviour of objects. A trace denotes a sequence of events which indicate sending and receiving of messages. It provides a state-less abstraction of the behaviour of an object.

The assertion language for describing properties of traces is an order-sorted logic with equality. It includes the elementary data types of the underlying programming language, for example, integers and booleans.

The names of operations specified in interfaces are only used in the definition of the additional abstract data type of *events*. Each event e has the following attributes: $e.\,\mathrm{name}$ denotes the name of the operation of the event, $e.\,\mathrm{sender}$ its sender, $e.\,\mathrm{receiver}$ its receiver, $e.\,\mathrm{args}$ its sequence of arguments, and $e.\,\mathrm{kind}$ indicates

- sending an asynchronous message (by *send*),
- reading an asynchronous message from the message queue (by *recv*),
- sending and receiving an operation call (by *call*), or
- sending and receiving a return of an operation call (by *return*).

The abstract data type of a trace is modelled by a sequence of events. We assume a distinguished variable θ of this type, which denotes a global trace of a system of objects.

The semantics of this language is standard, except for the interpretation of variables ranging over objects, whose domain are the objects occurring in θ.

Observe that in the proposed data type event we do not have a kind that describes the creation of objects. It is often not necessary to reason about object creation. Instead, one reasons about the interactions between objects. These interactions are constrained by the *object-structure* of the system, that is, the objects' knowledge of other objects. This knowledge may either be constrained using an invariant or inferred from the local traces by observing the use of other objects. Consequently, we abstract from object creation. In Section 6.5.1, however, we demonstrate that under certain circumstances it is necessary to observe object creation. In most cases the implicit knowledge of object creation contains complete information about the object structure and the acquaintance relation between objects.

We use the following notations:

The empty sequence is expressed by ϵ.

For $n \in \mathbb{N}$ define $\mathrm{below}(n) \stackrel{\mathrm{def}}{=} \{m \in \mathbb{N} \mid m < n\}$.

s_i refers to the $(i+1)$th element of the sequence s, whose length is expressed by $|s|$.

$s \preceq s'$ states that s is a prefix of s'.

For any sequence s and any natural number $n \leq |s|$ define $\mathrm{prefix}(s, n)$ to be the prefix of s with length n, that is, $\mathrm{prefix}(s, n) \preceq s$ and $|\,\mathrm{prefix}(s, n)| = n$.

Analogously, we define $\mathrm{suffix}(s, n)$ to be the suffix of s starting at position n.

$s \downarrow S$ expresses the *projection* on S, that is, the largest subsequence of s over elements from S. We also write $s \downarrow e$ instead of $s \downarrow \{e\}$.

A trace θ is called *local to an object o*, if o is the sender of any send event and the receiver of any read event, that is,

$$\mathrm{local}(\theta, o) \stackrel{\mathrm{def}}{=} \theta \downarrow \{e \mid \varphi(e)\} \quad ,$$

where $\varphi(e)$ is:

$$(e.\,\text{kind} = \mathit{send} \implies e.\,\text{sender} = o) \wedge (e.\,\text{kind} = \mathit{recv} \implies e.\,\text{receiver} = o) \quad .$$

For later use we list here some abstractions of a global trace θ, which are defined by suitable projection operations:

$$\begin{aligned}
\text{recvby}(\theta, o) &\stackrel{\text{def}}{=} \theta \downarrow \{e \mid e.\,\text{kind} = \mathit{recv} \wedge e.\,\text{receiver} = o\} \\
\text{recvfrom}(\theta, o) &\stackrel{\text{def}}{=} \theta \downarrow \{e \mid e.\,\text{kind} = \mathit{recv} \wedge e.\,\text{sender} = o\} \\
\text{sentby}(\theta, o) &\stackrel{\text{def}}{=} \theta \downarrow \{e \mid e.\,\text{kind} = \mathit{send} \wedge e.\,\text{sender} = o\} \\
\text{sentto}(\theta, o) &\stackrel{\text{def}}{=} \theta \downarrow \{e \mid e.\,\text{kind} = \mathit{send} \wedge e.\,\text{receiver} = o\}
\end{aligned}$$

The local assertion language is used for specifying local properties of the behaviour of an object and is obtained from the global assertion language by introducing a special logical variable *self*, denoting the object under consideration, and *interpreting* occurrences of θ as referring to $\text{local}(\theta, \mathit{self})$.

Additionally, we do not allow unbounded quantification over objects, but require that quantification over objects is bounded by the objects occurring in the trace.

Given a logical variable o, the substitution $[o/\mathit{self}]$ transforms a local assertion into a global one by replacing every occurrence of *self* by o and every occurrence of θ by $\text{local}(\theta, o)$.

6.4 Compositionality

After having introduced our trace logic, we describe our verification method. It is based on introducing local assertions $\mathcal{I}_c$ as interface invariants for each class $c \in C$, where C is the set of all classes occurring in the system, whereas the global assertion language is used to specify global properties ϕ of the communication network. Then the proof obligation for proving ϕ is expressed by the following formula, explained in a number of steps:

$$\mathfrak{A} \vdash \bigwedge_{c \in C} \forall z_c : \mathcal{I}[z_c/\mathit{self}] \implies \phi(\theta), \tag{6.1}$$

where $\mathfrak{A}$ axiomatises the theory of global traces, as described in Section 6.5, and where:

1. z_c expresses a variable ranging over instances of class c and the quantification, as described in the previous section, ranges over all instances of class c occurring in global trace θ, and

2. the substitution $[z_c/\mathit{self}]$ replaces every occurrence of *self* by z_c.

Observe that the substitution $[z_c/self]$ enforces in (6.1) that the local trace $\theta \downarrow self$ of the object denoted by z_c equals the projection of $\text{local}(\theta, z_c)$. Note that this substitution expresses compatibility between the local interface invariants in terms of the global trace θ.

Using this proof obligation we derive from the local specifications of each class a global property. This makes the proof method compositional.

6.5 Axiomatisation

In this section we describe the axioms of our trace logic, which can be used to prove properties of a system.

6.5.1 Observing Object Creation

In our trace logic we do *not* observe the creation of objects as such, because we claim that the creation of an object can be inferred from the global trace.

Intuitively, we infer that object o has created o' if it *reveals* o'. An object o' is revealed by o in a trace θ when o sends in θ a message to o' or it sends a message with o' as one of the parameter values without having received o' as a parameter of a communication record in θ before.[1]

$$\begin{aligned}\text{reveal}(\theta, o, o') \stackrel{\text{def}}{=} \exists i : \text{kind}(\theta_i) \neq \mathit{recv} \\ \wedge\ \text{sender}(\theta_i) = o \wedge (\text{receiver}(\theta_i) = o' \vee o' \in \theta_i.\,\text{args}) \\ \wedge\ \forall j < i : \theta_j.\,\text{receiver} = o \implies o' \notin \theta_j.\,\text{args} \quad (6.2)\end{aligned}$$

Define $\text{child}(\theta, o)$ as the set of children of o, where we call o' a *child* of o if o reveals o' in θ:

$$\text{child}(\theta, o) \stackrel{\text{def}}{=} \{o' \mid o \neq o' \wedge \text{reveal}(\theta, o, o')\}$$

The first axiom expresses that an object cannot have been revealed by two different objects in a trace θ.

Axiom 6.1. $\forall o, o' : o \neq o' \implies \text{child}(\theta, o) \cap \text{child}(\theta, o') = \emptyset$.

The second axiom of our trace logic prevents cycles in the chain of creation. To formalise this, we define the transitive closure of the child relation and call it offspring:

$$\text{offspring}(\theta, o) \stackrel{\text{def}}{=} \text{child}(\theta, o) \cup \{o' \mid \exists o'' \in \text{child}(\theta, o) : o' \in \text{offspring}(\theta, o'')\}$$

Axiom 6.2. $\forall o : o \notin \text{offspring}(o, \theta)$.

[1] We write $v \in \vec{v}$ if there is an index i such that $\vec{v}_i = v$.

Note that we do not require that every object has been created by another object. Such objects are called *root objects*. This implies that in a trace there can appear more than one root object. Such traces represent computations starting from a initial configuration containing possibly more than one root object.

Soundness

In this section we establish the soundness of our approach and derive precise conditions under which our abstraction is sound. We prove the soundness of our approach by establishing a Galois connection between traces with object creation and objects without object creation. We define the notion of Galois connection.

Definition 6.3. Let $(P, \leq)$ and $(Q, \sqsubseteq)$ be ordered sets. A pair (α, γ) of maps $\alpha : P \to Q$ and $\gamma : Q \to P$ is called a *Galois connection* between P and Q if for all $p \in P$ and $q \in Q$:

$$\alpha(p) \sqsubseteq q \Leftrightarrow p \leq \gamma(q) \tag{6.3}$$

◊

Following this definition we have to define two partially ordered sets of traces. Our trace logic is modelled by the set of traces without object creation. We order this set discretely, that is, the set of traces without object creation is ordered by equality.

Next, we introduce traces with object creation.

Definition 6.4. Let create(o, o') represent the observation that o creates o'. A trace θ is a *trace with object creation*, if every object occurring in a trace is created at most once and whenever o communicates with o', then o has received the identity of o' before or o has created o', that is, the following axioms hold:

$$\forall i, j : \exists o, o' : (\theta_i = \text{create}(o, o')) \wedge (\theta_j = \text{create}(o, o')) \implies (i = j) \tag{6.4}$$

$$\begin{aligned} \forall o : \forall i : (\theta_i.\,\text{kind} = \mathit{send} \vee \theta_i.\mathit{kind} = \mathit{call}) \implies \\ \forall o' \in \{\theta_i.\,\text{receiver}\} \cup \theta_i.\,\text{args} : o = o' \vee \\ \Big(\exists j < i : (\theta_j = \text{create}(o, o')) \vee \\ ((\theta_j.\,\text{kind} = \mathit{recv} \vee \theta_j.\,\text{kind} = \mathit{call}) \wedge \\ o = \theta_j.\,\text{receiver} \wedge o' \in \theta_j.\,\text{args})\Big) \end{aligned} \tag{6.5}$$

and

$$\forall o' : \exists i, o : \theta_i = \text{create}(o, o') \implies \forall j < i : o' \notin \theta_j \quad , \tag{6.6}$$

where $o' \notin \theta_j$ states that o' does not occur in any of the fields of θ_j. Additionally, we require that traces satisfy Axiom 6.2, that is, the relation of object creation, is acyclic. ◊

Equation (6.4) corresponds to Axiom 6.1, namely, that each object is created by at most one other object.

Equation (6.5) asserts that whenever an object reveals another object, then it has created this object before.

Finally, Equation (6.6) states that each object that is created by another object is actually created before its first activity.

We observe the following property:

Remark 6.1. For any trace θ with object creation, if $\text{reveal}(\theta, o, o')$ holds then there is an index i such that $\theta_i = \text{create}(o, o')$ and $\neg\, \text{reveal}(\text{prefix}(\theta, i), o, o')$.

Equation (6.5) is stating the same fact as (6.2) states.

For convenience, we write

$$\text{nocreate}(\theta) \stackrel{\text{def}}{=} \theta \downarrow \{e \mid e.\,\text{kind} \neq \text{create}\}$$

and

$$\text{created}(\theta) = \{o \mid \exists i : \exists o' : \theta_i = \text{create}(o', o)\} \quad .$$

Now we define a partial order on traces with object creation. The intention is to consider a trace "larger" (more abstract) if it is "lazier" in creating new objects (that is, it creates objects later).

In the next definition we use the fact that projection can be expressed using a largest strictly monotonically increasing function f. We use the notation f^- for the inverse of a function.

Definition 6.5. Let θ and θ' be traces with object creation. We say that θ creates objects *more lazily* than θ', written $\theta' \leq \theta$, if and only if

$$\text{nocreate}(\theta) = \text{nocreate}(\theta') \tag{6.7}$$

and if f is the projection function from θ to $\text{nocreate}(\theta)$, g the projection function from θ' to $\text{nocreate}(\theta')$, then for all $i \in \text{below}(|\,\text{nocreate}(\theta)|)$

$$\text{created}(\text{prefix}(\theta, f^-(i))) \subseteq \text{created}(\text{prefix}(\theta', g^-(i))) \tag{6.8}$$

and

$$\text{created}(\theta) \subseteq \text{created}(\theta') \quad . \tag{6.9}$$

◊

Equation (6.7) states that we only order traces which represent the same communication behaviour.

Equation (6.8) allows the eager trace θ' to create more objects and to create them earlier. By Equation (6.5) these additionally created objects in θ' are redundant, because no messages are sent to them and their values are never communicated.

Equation (6.8) does not consider the case that the traces θ and θ' have suffixes which only consist of object creations, because the projection functions have proper communications in domain range and these suffixes do not occur in the range of the projection functions. Therefore, Equation (6.9) is required to assert that θ creates less objects or creates objects later than θ' in this suffix.

One important property of Definition 6.5 is that traces in which objects are created consecutively, but in a different order, are equivalent. In Equations (6.8) and (6.9) we compare the objects in the prefixes which represent the same communication behaviour.

When reconstructing a trace with object creation and we encounter a send event where more than one object is revealed, we have to guess an order, in which these objects have been created (as done in Equation (6.10)).

Because the order is not determined, we find traces θ and θ' with $\theta \neq \theta'$, $\theta \leq \theta'$ and $\theta' \leq \theta$. Consequently, we need a different notion of equivalence. Therefore, we write $\theta \gtrless \theta'$ if $\theta \leq \theta'$ and $\theta' \leq \theta$. One can easily establish the following remark, exploiting the fact that $\theta \leq \theta'$ and $\theta' \leq \theta$ expresses that θ and θ' differ in the order of consecutive create events:

Remark 6.2. $\gtrless$ is an equivalence relation.

Next, we establish that $\leq$ as defined in Definition 6.4 is indeed a partial order:

Lemma 6.6. *Let $\leq$ as defined in Definition 6.5. Then $\leq$ is a partial order on traces with object-creation modulo $\gtrless$.*

Proof. Apparently, $\leq$ is reflexive.

We prove that $\leq$ is transitive: Let $\theta \leq \theta'$ and $\theta' \leq \theta''$. From (6.7) we conclude $\text{nocreate}(\theta) = \text{nocreate}(\theta')$ and $\text{nocreate}(\theta') = \text{nocreate}(\theta'')$. Consequently, $\text{nocreate}(\theta) = \text{nocreate}(\theta'')$.

Let $i \in \text{below}(|\text{nocreate}(\theta)|)$, f the function projecting θ onto $\text{nocreate}(\theta)$, g the function projecting θ' onto $\text{nocreate}(\theta')$, and h the function projecting θ' onto $\text{nocreate}(\theta')$. Then Equation (6.8) implies $\text{created}(\text{prefix}(\theta, f^{-}(i))) \subseteq \text{created}(\text{prefix}(\theta'', h^{-}(i)))$.

From the assumptions we conclude $\theta \leq \theta' \implies \text{created}(\theta') \subseteq \text{created}(\theta)$ and $\theta' \leq \theta'' \implies \text{created}(\theta'') \subseteq \text{created}(\theta')$. The transitivity of $\leq$ implies $\text{created}(\theta'') \subseteq \text{created}(\theta)$.

Finally, we prove antisymmetry modulo permutations of consecutive create events. Let θ, θ' such that $\theta \leq \theta'$ and $\theta' \leq \theta$. From (6.7) we conclude that θ and θ' represent the same behaviour. Now we show that each event, except consecutive create events, occur

on the same position. Let $i \in \text{below}(|\,\text{nocreate}(\theta)|)$, f the function projecting θ onto $\text{nocreate}(\theta)$, and g the function projecting θ' onto $\text{nocreate}(\theta')$. Again, we conclude

$$\text{created}(\text{prefix}(\theta, f^{-}(i))) = \text{created}(\text{prefix}(\theta', g^{-}(i)))$$

from (6.8). It follows that θ and θ' create the same objects between the same events which are not a create event. Only a different order of these creates are allowed.

Analogous reasoning establishes Equation (6.9). □

As required by Definition 6.3 we have now defined two partially ordered sets, namely the set of traces without object creation, which we order discretely, and the set of traces with object creations ordered by $\leq$. Next we have to define two functions α and γ to establish the Galois connections. As α we use the function nocreate. Gamma is defined next. We need the following notations:

$$\text{start}(\theta) \stackrel{\text{def}}{=} \theta_0 \cdots \theta_{|\theta|-2}$$

and

$$\text{last}(\theta) \stackrel{\text{def}}{=} \theta_{|\theta|-1} \quad .$$

The operator $\cdot$ expresses concatenation of finite sequences. If no ambiguity arises, we treat elements as sequences of length 1 for the purpose of concatenation.

As γ we define a function that creates objects as late as possible. Before we define this function, we need some auxiliary definitions. Let θ be a trace with object creation, o and o' objects, and $i = \max\{j \mid o' \notin \theta_j\}$ if o' occurs in θ and $i = |\theta|$ if o' does not occur in θ. Then the function insert inserts a create-event into the trace before the *first* occurrence of o', or appends the event if o' does not occur in θ:

$$\text{insert}(\theta, o, o') \stackrel{\text{def}}{=} \text{prefix}(\theta, i) \cdot \text{create}(o, o') \cdot \text{suffix}(\theta, i)$$

Observe that it is possible to create any number of objects during one observation of a trace without object creation, as they all can be revealed by passing them as parameters. Therefore, we extend insert to inserting a set O of objects created by o as follows:

$$\text{insert}(\theta, o, O) \stackrel{\text{def}}{=} \begin{cases} \theta & \text{if } O = \emptyset \\ \text{insert}(\theta, o, o') & \text{if } O = \{o'\} \\ \text{insert}(\text{insert}(\theta, o, O \setminus \{o'\}), o, o') & \text{if } |O| > 1 \text{ and } o' \in O. \end{cases}$$

This extended "function" insert is actually not a function, because the order in which the elements of O are inserted is not determined. But the order in which the create-events are inserted is not relevant and these different orders are considered equal by $\gtrless$, as shown by the following Lemma:

Lemma 6.7. *Let θ express a trace with object creation, o some object and O a set of objects. Then all results of* $\mathrm{insert}(\theta, o, O)$ *are in an $\gtrless$-equivalence class.*

We define γ inductively as follows:

$$\gamma(\theta) \stackrel{\text{def}}{=} \begin{cases} \epsilon & \text{if } \theta = \epsilon \\ \gamma(\mathrm{start}(\theta)) \cdot \mathrm{last}(\theta) & \text{if } \mathrm{child}(\theta, o) = \mathrm{child}(\mathrm{start}(\theta), o) \\ \mathrm{insert}(\gamma(\mathrm{start}(\theta)), o, O) \cdot \mathrm{last}(\theta) & \text{otherwise,} \end{cases} \tag{6.10}$$

where $o = \mathrm{last}(\theta).\,\mathrm{sender}$ and $O = \mathrm{child}(\theta, o) \setminus \mathrm{child}(\mathrm{start}(\theta))$.

Before we prove the main result of this section, we first have to establish that the result of γ is indeed a valid trace with object creation.

Lemma 6.8. *Let θ be a trace without object-creation. Then $\gamma(\theta)$ is a trace with object creation, that is, it satisfies Definition 6.4.*

Proof. By induction on θ. Let θ be a trace without object creation.

Case $\theta = \epsilon$. Then $\gamma(\theta) = \epsilon$ and ϵ satisfies Definition 6.4.

Case $\theta \neq \epsilon$. Assume as an induction hypothesis $\gamma(\mathrm{start}(\theta))$ satisfies Definition 6.4. Let, for the remainder of the proof, $o = \mathrm{last}(\theta).\,\mathrm{sender}$. We distinguish two sub-cases:

If $\mathrm{child}(\theta, o) = \mathrm{child}(\mathrm{start}(\theta), o)$, then the last communication did not create any new objects. By induction hypothesis, we know that $\gamma(\mathrm{start}(\theta))$ satisfies Definition 6.4. Because $\mathrm{child}(\theta, o) = \mathrm{child}(\mathrm{start}(\theta), o)$ also $\gamma(\mathrm{start}(\theta)) \cdot \mathrm{last}(\theta)$ satisfies Definition 6.4.

If $\mathrm{child}(\theta, o) \neq \mathrm{child}(\mathrm{start}(\theta), o)$ holds, then $\mathrm{child}(\mathrm{start}(\theta), o) \subset \mathrm{child}(\theta), o)$ holds, too. Then $O = \mathrm{child}(\theta, o) \setminus \mathrm{child}(\mathrm{start}(\theta), o)$. Observe that O is not empty and finite. Next, we prove that $\mathrm{insert}(\gamma(\mathrm{start}(\theta)), o, O)$ satisfies Definition 6.4 by induction on O.

If $O = \emptyset$ then $\mathrm{insert}(\gamma(\mathrm{start}(\theta)), o, \emptyset) = \gamma(\mathrm{start}(\theta))$, which, by the first induction hypothesis, satisfies Definition 6.4.

Assume $O \neq \emptyset$. Let $o' \in O$, let $\theta' = \mathrm{insert}(\gamma(\mathrm{start}(\theta)), o, O \setminus \{o'\})$, and assume as induction hypothesis that θ' satisfies Definition 6.4. To prove: $\mathrm{insert}(\theta', o, o')$ satisfies Definition 6.4. From the assumption that o' is an object revealed by o and the creation of o' has not been inserted into θ' we conclude there is no o'' such that $\mathrm{create}(o'', o')$ occurs in θ'.

Let i be the largest number such that o' does not occur in $\mathrm{prefix}(\theta', i)$. If θ'_i exists, then θ'_i is the first occurrence of o', otherwise it is $\mathrm{last}(\theta)$. Therefore, in the trace $\mathrm{prefix}(\theta', i) \cdot \mathrm{create}(o, o') \cdot \mathrm{suffix}(\theta', i)$ only contains one $\mathrm{create}(o, o')$, as required by Equation (6.4) and since o' does not occur in $\mathrm{prefix}(\theta', i)$, $\mathrm{prefix}(\theta', i) \cdot \mathrm{create}(o, o') \cdot \mathrm{suffix}(\theta', i)$ also satisfies Equation (6.5). Consequently, $\mathrm{insert}(\theta', o, o')$ satisfies Definition 6.4. □

This lemma establishes the type-correctness of the function γ. Moreover, it demonstrates the necessary conditions under which one may abstract from object creation in a trace-based theory. The function γ is defined in terms of a function insert which inserts the necessary create events into a trace. The function does not insert the create

observation before the object which has to be created is revealed. The reason for this is that the created object is *active*, that is, it has its own thread of control and starts communicating with other objects as soon as it has been created.

In our theory passing parameters to an objects constructor method is modelled as ordinary communication. In this case, the newly created object waits for its creator to call the constructor method. But it need not wait but can, after initialising into a default state, create its own objects and communicate with them. In this case, an object may be revealed *after* it has sent messages. Within the prefix of a trace before the object has been revealed, it appears to be a root-object.

Consider a *set* of traces without object creation Θ characterising all computations of a system. Furthermore, assume that Θ is prefix-closed, as required by the theory of Zwiers [158]. Then the set of traces with object creation $\{\gamma(\theta) \mid \theta \in \Theta\}$ is generally not prefix-closed, because in the prefixes of traces in which an active object has been created, that object appears to be a root-object. For systems with active objects specified in our trace logic where specifications have to be prefix-closed, one either has to make sure that object-creation does not matter or one has to observe it, because it *cannot* be reconstructed using the function γ.

These considerations also imply that if we *require* that specifications are prefix-closed, these results could not have been established. For active objects one has often to know the continuation of a trace in order to decide whether an object has been created by another one.

Lemma 6.9. *For all traces θ without object creation* $\mathrm{nocreate}(\gamma(\theta)) = \theta$ *holds.*

Now we prove the main theorem of this section:

Theorem 6.10. *Let $\alpha =$* nocreate *and γ as defined in* (6.10)*. Then (α, γ) is a Galois connection between the set of traces without object creation and the set of traces with object creation.*

Proof. Case $\alpha(\theta) = \hat{\theta} \implies \theta \leq \gamma(\hat{\theta})$: Let θ be a trace with object creation and $\hat{\theta}$ a trace without object creation such that $\alpha(\theta) = \hat{\theta}$. Using Lemma 6.9 we obtain $\alpha(\theta) = \mathrm{nocreate}(\gamma(\hat{\theta}))$. It remains to prove (6.8). From the assumption $\hat{\theta} = \alpha(\theta)$ we find a projection function $f : \mathrm{dom}(\theta) \longrightarrow \mathrm{dom}(\hat{\theta})$. Let g be the projecting function from $\gamma(\hat{\theta})$ to $\hat{\theta}$. Let $i \in \mathrm{below}(|\hat{\theta}|)$. Assume there exists a $p \in \mathrm{created}(\mathrm{prefix}(\gamma(\hat{\theta}), g^-(i)))$ such that $p \notin \mathrm{created}(\mathrm{prefix}(\theta, f^-(i)))$. Then we find a smallest i' such that $p \in \mathrm{created}(\mathrm{prefix}(\gamma(\hat{\theta}), g^-(i')))$ and, as a consequence of (6.10), $\hat{\theta}_{i'}$ is the first event where p is revealed. Because $\hat{\theta} = \alpha(\theta) = \mathrm{nocreate}(\theta)$ we know that $\alpha(\theta)_{i'}$ is also the first event where p is revealed. However, we assumed $p \notin \mathrm{created}(\mathrm{prefix}(\theta, f^-(i)))$, which contradicts Definition 6.4. Therefore, there is no such p, and (6.8) holds. Equation (6.9) follows from the fact that $\gamma(\hat{\theta})$ never ends in a create event.

Case $\theta \leq \gamma(\hat{\theta}) \implies \alpha(\theta) = \hat{\theta}$: Let θ be a trace with object creation and $\hat{\theta}$ a trace without object creation such that $\theta \leq \gamma(\hat{\theta})$. Therefore $\alpha(\theta) = \alpha(\gamma(\hat{\theta}))$ by (6.7). Now

observe that $\alpha(\theta) = \text{nocreate}(\theta)$, from which we conclude $\alpha(\theta) = \alpha(\gamma(\hat{\theta}))$. Using Lemma 6.9 we finish the proof. □

The existence of a Galois connection between traces with object creation and without object creation asserts that properties specified on traces without object creation also hold for traces with object creation obtained by applying the function γ and traces which are more eager, that is, where object creation is observed earlier, than the traces obtained from γ. This also means that one can, if memory is not limited, abstract from object creation in specifications. In general, creating an object is not part of the behaviour. Similar ideas were presented by Cousot and Cousot in [36] and by Dams in [40] in the analysis of programs. The idea of lazy object creation has also been applied by Ábrahám et al in, for example, [2] for establishing a fully abstract semantics of object-oriented programs.

6.5.2 Communication Mechanisms

We assume that all asynchronous messages are received in the event queue in a first-in-first-out order. This is expressed by Axiom 6.11.

Axiom 6.11. $\forall o : \zeta(\text{recvby}(\theta, o)) \preceq \zeta(\text{sentto}(\theta, o))$, where ζ is the function which removes kind from all communication records.

Note that any other kind of asynchronous communication could be adopted as well.

Finally, observe that synchronous communication is modelled by the definition of local traces as projections of the global trace (cf. [158]).

Axiom 6.12. $\forall o : \zeta(\text{recvby}(\theta, o)) = \zeta(\text{sentto}(\theta, o))$, where ζ is the function which removes kind from all communication records.

We also assume that synchronous communication is non-reentrant, meaning that, after an object has accepted a call, it first has to return before accepting another call or receiving another message. This is expressed by the next axiom:[2]

Axiom 6.13.

$$\begin{aligned}\forall o : \forall i : \theta_i.\,\text{kind} = \mathit{return} \wedge \theta_i.\,\text{sender} = o \implies \\ \exists j < i : \theta_j.\,\text{kind} = \mathit{call} \wedge \theta_j.\,\text{receiver} = o \wedge \\ \theta_j.\,\text{sender} = \theta_i.\,\text{receiver} \wedge \theta_j.\,\text{name} = \theta_i.\,\text{name} \wedge \\ \exists v : \theta_j.\,\text{args} = \theta_i.\,\text{args} \cdot v \wedge \\ \forall k : j < k \wedge k \leq i : (\theta_k.\,\text{kind} = \mathit{call} \vee \theta_k.\,\text{kind} = \mathit{recv}) \implies \theta_k.\,\text{receiver} \neq o\end{aligned}$$

[2] This axiom expresses a communication mechanism used by UML state machines, as defined by the Object Management Group [111]. To obtain reentrant operation calls, this axiom has to be adapted.

6.6 Sieve of Eratosthenes

In this section we specify the "Sieve of Eratosthenes" and prove its correctness. The Sieve of Eratosthenes is an efficient algorithm for finding all prime numbers. The idea is to have a generator which sends the natural numbers starting with 2 to a sieve object. A sieve object decides, depending on the first number it has received, whether it may be a prime or not. If the number received is divisible by the first number it has received, then it is a composite number and cannot be a prime number. If the sieve object determines that the received number is not divisible by the first number received, it sends the number to the next sieve object.

More precisely, we have two classes: *Generator* and *Sieve*. Instances g of class *Generator* create exactly one instance of class *Sieve*, see (6.11) below, and then send it the increasing sequence $2, 3, 4, \ldots$ of natural numbers as specified by (6.12) below, identifying functions over $\mathbb{N}$ with sequences. Instances s of class Sieve also create exactly one instance of class Sieve by (6.11) and then "sift" the sequence of numbers they receive by (6.13), where this sifting process is recursively defined in (6.14).

$$\forall o : \exists o' : \mathrm{child}(\theta, o) \subseteq \{o'\} \tag{6.11}$$

$$\mathrm{sentby}(\theta, g).\,\mathrm{args} \preceq \lambda n.n + 2 \tag{6.12}$$

$$\mathrm{sentby}(\theta, o).\,\mathrm{args} \preceq \mathit{Sieve}(\mathrm{recvby}(\theta, o).\,\mathrm{args}, \mathrm{recvby}(\theta, o)_0.\,\mathrm{args}) \tag{6.13}$$

For s a sequence over $\mathbb{N}$ and $n \in \mathbb{N}$ define

$$\mathrm{sift}(s, n) \stackrel{\mathrm{def}}{=} \begin{cases} \epsilon & \text{if } s = \epsilon \\ \mathrm{sift}(\mathrm{tail}(s), n) & \text{if } n \mid \mathrm{head}(s) \\ \mathrm{head}(s) \cdot \mathrm{sift}(\mathrm{tail}(s), n) & \text{if } n \nmid \mathrm{head}(s), \end{cases} \tag{6.14}$$

where $n \mid s$ states that s divides n and $n \nmid s$ states that s does not divide n.

Here the class invariant $\mathcal{I}_{\mathit{Generator}}$ of *Generator* is the conjunction of (6.11) and (6.12) and $\mathcal{I}_{\mathit{Sieve}}$ is the conjunction of (6.11) and (6.13).

For each instance g of Generator we find a sequence of sieve instances which each have received a prime number as their first value. This pipe structure p denotes a sequence of objects determined by the global trace θ. It is inductively defined by the predicate

$$\Pi(\theta, p) \stackrel{\mathrm{def}}{=} \begin{cases} \mathrm{true} & \text{if } |p| \leq 1, \\ \mathrm{head}(\mathrm{tail}(p)) \in \mathrm{child}(\theta, \mathrm{head}(p)) \wedge \Pi(\theta, \mathrm{tail}(p)) & \text{otherwise,} \end{cases}$$

expressing that every object in the pipe only sends messages to its child, being its successor. If an object sends a message to objects other than its successor, the object has either received another object identity, which contradicts (6.12) and (6.13), because only the sending of numbers is allowed, or it has created a second child, which contradicts (6.11). Observe that the first object p_0 in p denotes the generator.

As required in (6.1) we now have the local invariants $\mathcal{I}_{Generator}$ (the conjunction of (6.11) and (6.12)) and $\mathcal{I}_{Sieve}$ (the conjunction of (6.11) and (6.13)), as well as the the global sifting property $\phi(\theta)$ defined in (6.15) below. Validity of the resulting formula is formulated in Theorem 6.14 and proved later. Observe that $\Pi(\theta, p)$ is not an additional assumption. As explained above, its validity can be derived from (6.11–6.13).

Theorem 6.14. *For any global trace θ satisfying (6.11–6.13) with p ranging over sequences of objects satisfying $\Pi(\theta, p)$ one has*

$$\forall i \in \text{below}(|p| - 1) : \text{sentby}(\theta, p_{i+1}).\,\text{args} \leq \text{sift}(\text{sentby}(\theta, p_i).\,\text{args}, \text{sentby}(\theta, p_i)_0.\,\text{args}) \quad (6.15)$$

Lemma 6.15. *For any sequences θ and θ' with $\theta \preceq \theta'$ and any n we have* $\text{sift}(\theta, n) \preceq \text{sift}(\theta', n)$.

Proof. By induction on θ and θ' (cf. also [81]). □

Next define

$$O(\theta) \stackrel{\text{def}}{=} \{o \mid \exists i : \text{sender}(\theta_i) = o \vee \text{receiver}(\theta_i) = o \vee o \in \theta_i.\,\text{args}\} \quad .$$

Lemma 6.16. *Let θ be a trace of the sieve system and p a sequence of objects such that $\Pi(\theta, p)$ holds. Then* $\text{sentto}(\theta, p_{i+1}) = \text{sentby}(\theta, p_i)$ *for all* $i \in \text{below}(|p| - 1)$.

Proof. By induction on the length of the trace θ.

If $|\theta| = 0$ then $\text{sentto}(\theta, p_{i+1}) = \epsilon$ and $\text{sentby}(\theta, p_i) = \epsilon$.

Suppose $|\theta| > 0$. With the assumptions $\Pi(\text{start}(\theta), p)$ and $\text{sentto}(\text{start}(\theta), p_{i+1}) = \text{sentby}(\text{start}(\theta), p_i)$ we distinguish two sub-cases:

Case $p_{i+1} \in O(\text{start}(\theta))$: The induction hypothesis states $p_{i+1} \in \text{child}(\text{start}(\theta), p_i)$.

Because $\text{last}(\theta).\,\text{kind} = \textit{recv}$ and because sentto and sentby observe only *send* events and no *recv* events, these equalities hold:

$$\text{sentto}(\theta, p_{i+1}) = \text{sentto}(\text{start}(\theta), p_{i+1})$$
$$\text{sentby}(\text{start}(\theta), p_i) = \text{sentby}(\theta, p_i)$$

Again, we distinguish two sub-cases:

Case $\text{last}(\theta).\,\text{kind} = \textit{send}$ and $\text{last}(\theta).\,\text{sender} \neq p_i$. From the fact $\text{child}(\theta, p_i) = \{p_{i+1}\}$, $\Pi(\text{start}(\theta), p)$, and from

$$\text{sentto}(\theta, p_{i+1}) = \text{sentto}(\text{start}(\theta), p_{i+1})$$

and, therefore,

$$\text{sentby}(\text{start}(\theta), p_i) = \text{sentby}(\theta, p_i) \quad ,$$

we conclude $\text{last}(\theta).\,\text{receiver} \neq p_{i+1}$.

If $\text{last}(\theta).\,\text{kind} = \mathit{send}$ and $\text{last}(\theta).\,\text{sender} = p_i$, then $\text{child}(\theta, p_i) = \{p_{i+1}\}$ (again, because $\Pi(\text{start}(\theta), p)$ holds) and therefore

$$\text{sentto}(\theta, p_{i+1}) = \text{sentto}(\text{start}(\theta), p_{i+1}) \cdot \mathit{send}(p_i, p_{i+1}, e, v)$$

and

$$\text{sentby}(\text{start}(\theta), p_i) \cdot \mathit{send}(p_i, p_{i+1}, e, v) = \text{sentby}(\theta, p_i) \quad ,$$

which proves the claim for this case.

Case $p_{i+1} \notin \mathcal{O}(\text{start}(\theta))$: The induction hypothesis states $p_{i+1} \notin \text{child}(\text{start}(\theta), p_i)$. Because of $p_{i+1} \in \text{child}(\theta, p_i)$, the last observation in θ is sending a message from p_i to p_{i+1}, that is, $\text{last}(\theta) = \mathit{send}(p_i, p_{i+1}, e, v)$ for some value v. Using the induction hypothesis and the definitions of sentto and sentby we have

$$\begin{aligned}\text{sentto}(\theta, p_{i+1}) = \text{sentto}(\text{start}(\theta), p_{i+1}) \cdot \mathit{send}(p_i, p_{i+1}, e, v) = \\ \text{sentby}(\text{start}(\theta), p_i) \cdot \mathit{send}(p_i, p_{i+1}, e, v) = \text{sentby}(\theta, p_i)\end{aligned}$$

In any case the claim holds. □

Proof. [Proof of Theorem 6.14] By induction on p.

Let θ be a trace and p a sequence of objects such that $\Pi(\theta, p)$ holds.

- If $|p| \leq 1$, that is, if only p_0 exists, the claim is trivially true.
- Assume $|p| > 1$, $p_1 \in \text{child}(\theta, p_0)$, and, as an induction hypothesis, that Theorem 6.14 holds for $\text{tail}(p)$.

  To prove:

  $$\text{sentby}(\theta, p_1).\,\text{args} \preceq \text{sift}(\text{sentby}(\theta, p_0).\,\text{args}, \text{sentby}(\theta, p_0)_0.\,\text{args}).$$

  - If $\text{sentby}(\theta, p_1) = \epsilon$, then the claim is trivially true.
  - Assume that $\text{sentby}(\theta, p_1) \neq \epsilon$. Then $\text{sentby}(\theta, p_1)_0$ and $\text{recvby}(\theta, p_1)_0$ are defined. Lemma 6.16 implies $\text{sentto}(\theta, p_1) = \text{sentby}(\theta, p_0)$ and Axiom 6.11 implies

    $$\text{recvby}(\theta, p_1) \preceq \text{sentto}(\theta, p_1),$$

    resulting in

    $$\text{recvby}(\theta, p_1) \preceq \text{sentby}(\theta, p_0).$$

Use Lemma 6.15, Axiom 6.11, and $\text{recvby}(\theta, p_1)(0) = \text{sentby}(\theta, p_0)(0)$ to obtain

$$\text{sift}(\text{recvby}(\theta, p_1).\,\text{args}, \text{recvby}(\theta, p_1)_0.\,\text{args}) \preceq \text{sift}(\text{sentby}(\theta, p_0).\,\text{args}, \text{sentby}(\theta, p_0)_0.\,\text{args})$$

Use the transitivity of $\preceq$ and Formula (6.13) as

$$\text{sentby}(\theta, p_1).\,\text{args} \preceq \text{sift}(\text{recvby}(\theta, p_1).\,\text{args}, \text{recvby}(\theta, p_1)_0.\,\text{args})$$

to obtain

$$\text{sentby}(\theta, p_1).\,\text{args} \preceq \text{sift}(\text{sentby}(\theta, p_0).\,\text{args}, \text{sentby}(\theta, p_0)_0.\,\text{args}).$$

□

6.7 Related Work

The method described in this paper represents an extension of the method described by Jonsson in [76] on asynchronous buffered communication to object creation and synchronous message passing in the context of a rendez-vous.

An early reference to some of the techniques we use is Ole-Johan Dahl's "Can Program Proving be Made Practical" (cf. [38]), because this and his work use finite traces for object specification and induction as the prime proof method.

Compatibility, as used in Section 6.4, was introduced by Soundararajan in [144]. In this paper, the existence of a global trace satisfying the interface specification expresses that two objects are compatible, that is, can be composed. The existence of such a trace is not mandatory in our setting, because the global property is proved for all global traces, and is vacuously true, if no global trace satisfying all interface specifications exist.

Instead of predicates in a trace logic, interfaces can also be expressed in temporal logic, as proposed by Lamport in [89], or finite automata, as proposed by Alfaro et al in [41]. Wolfgang Thomas [147] proves that a language based on first order logic is more expressive than a language based on finite automata. For example, a stack discipline, as it occurs in call-return pairs of operation calls, cannot be expressed in this automata setting.

The method presented in this thesis can straightforwardly by applied to the extension of OCL defined in Chapter 5. Known approaches towards tool-based (automatic) verification for object creation concern bounded creation of objects and are implementation dependent, that is, not compositional, as demonstrated by the work of Damm et al [39] and of Ober et al [102].

6.8 Conclusion and Future Work

In this paper we have presented a trace-based specification method for object-oriented programs with object creation. We have proved that not observing object creation but inferring object creation from the trace is a feasible abstraction. Most behavioural specifications are not necessarily concerned with object creation but with the exchange of messages.

We have made explicit that a specification of the behaviour of an object does not only involve the services it provides, but also the context in which this object operates, and the services it requires. Besides this static information, the specification of an object is a contract or a protocol of the behaviour between objects. These are expressed by interface invariants.

We have proposed a proof rule for composing interface specifications and deriving global properties. This allows the verification of systems during early stages of design, where an implementation of each object or class is not yet known.

We have applied our trace logic to a case study: the sieve of Eratosthenes. Our method seems to be particularly well suited for stream-processing applications, as demonstrated by our case study.