# State-based Object Models Are More Abstract Than Trace-based Models

## Towards a Unified Specification Framework

Ilham W. Kurnia, Arnd Poetzsch-Heffter, Yannick Welsch*

University of Kaiserslautern, Germany
ilham,poetzsch,welsch@cs.uni-kl.de

**Abstract.** The literature distinguishes between trace-based and state-based specification techniques for object-oriented components. Trace-based specifications describe behavior in terms of the message histories of components, while state-based techniques explain component behavior in terms of states. The latter define how the state is changed by method calls and what is returned as a result. The state space is either abstract or concrete. Abstract states are used to model the behavior without referring to the implementation. Concrete states are expressed by the underlying implementation. State-based specifications are usually described in terms of pre- and postconditions of methods.
In this paper, we investigate the relationship between trace-based specifications and specifications based on abstract states for sequential, object-based components. We first generalize state-based techniques so that they can handle callbacks. Then, we develop formal models for trace-based and state-based specifications and show that every trace-based model can be canonically represented as a state-based model. Adapting notions from process simulation, we define an abstraction relation between two state-based models allowing their comparison. In particular, state-based models are more abstract than trace-based models. We also show that there exist most abstract models. The developed framework is illustrated by a subject component of the Subject-Observer Pattern.

## 1 Introduction

Objects or, more generally, object-based components communicate with their environment via messages. Usually, the reaction to an incoming message depends on the state of the object or component. To avoid uncontrolled access and to achieve implementation-independency, it is an accepted principle to encapsulate the *concrete state* of the implementation. In particular, the concrete state should not be exposed in specifications of classes and components (see, e.g., [1, §1.3]). Two different kinds of implementation-independent specification techniques have been investigated in the literature. In so-called *model-* or *abstract state-based* specifications, behavior is explained based on a model or space of abstract states

* This work is partly funded by the EU project FP7-231620 HATS: Highly Adaptable and Trustworthy Software using Formal Methods

(see, e.g., [2]). The specification expresses the result values and changes of the abstract state in reaction to an incoming message. In the following, we will simply call these specifications *state-based*. In *trace-based* specifications, behavior is explained with respect to the history of messages an object or component has seen. In both cases, we say that a specification describes a *model* of the specified component. Models can be considered as the semantics of specifications. They will be formalized as transitions systems.

State-based specifications are closer to the implementation, can directly be complemented with invariants – which is important for verifying implementations –, and are supported by a set of well-developed specification constructs going far beyond the basic pre- and postconditions (see, e.g., [3,4]). On the other hand, trace-based specifications have a natural link to semantics (see [5]), avoid the design of an abstract state space, and can more easily deal with callbacks and concurrency (see, e.g., [6, §5]). Furthermore, invariants may also be specified to restrict, e.g., the structure of the allowed traces.

The long-term goal of our research is to work out the relationship between state- and trace-based specifications to combine the best of the two techniques. In this paper we investigate, as an initial step, specifications of sequential components with possible callback behavior. We consider specifications of object-based components where a component description consists of one or more classes. A (runtime) component is created by instantiating a class. Internally, the component can create further objects and expose these objects to the environment. To keep the presentation focused, we do not discuss concurrency and inheritance. However, we believe that our results can be generalized to such settings.

*Contributions.* To handle callback scenarios, we first generalize state-based specifications. In addition to JML-like pre- and postconditions of methods calls [4], we also allow to express what a component ensures when a call leaves the component and what it requires when the call is completed (returns). In this paper, this generalization is only to ease comparison between the different models. We illustrate the approach for a simple version of a `Subject` component following the Subject-Observer Pattern (SOP) [7].

The central contribution of the paper is to relate trace-based and state-based models in a formal way. In particular, we show that every trace-based model has an equivalent state-based model. We define an abstraction relation between two models which allows to compare two models. As results from the process simulation theory, we derive that there exist most abstract models. Knowing that a model is most abstract guarantees that it does not contain redundant states. Thus, the knowledge can be used as a guidance to remove redundancy from specifications. As an example, we show that the state-based model of the presented subject component example is most abstract.

*Paper structure.* Section 2 presents the specifications of the `Subject` component of the SOP in trace- and state-based manners. Sections 3 and 4 formally define our trace- and state-based models, respectively, and show how a trace-based model can be represented as a state-based model. Section 5 defines the

```
interface Observer {
  void notify(State s);
}

class Subject {
  Observer o1, o2;

  Subject(Observer o1, Observer o2) {
    this.o1 = o1; this.o2 = o2;
  }

  void update(State s) {
    o1.notify(s);
    o2.notify(s);
  }
}
```

**Fig. 1.** A `Subject` implementation

abstraction and states that state-based models are abstractions of trace-based models. We then relate this work to others and give some directions regarding future work.

## 2 Motivating Example

In Fig. 1, we give an implementation of the SOP. For simplicity, the `Subject` class stores exactly two observers. Each time the subject is updated with a new state, this state is propagated to the observers in a fixed order (i.e., `o1` then `o2`). In the presence of callbacks, there can be a sequence of notifications to `o1` before `o2` is notified. For example, the observer `o1`, upon being notified, may trigger a new update of the subject. The implementation does not guarantee that the second observer receives the states in the same order as the first observer.

The behavior of the SOP can be described using state- or trace-based specifications. For both kind of specification techniques, we need to address what the input and output of an object-oriented component are.

**Input/Output.** In general, input/output occurs at the places where the control flow enters or leaves the component. In our example, control flow can enter the component when the constructor or the `update` method is called by an object (outside of the component/from the *environment*). Control flow may leave the component when the constructor or the `update` method returns. One must also note that control flow leaves the component when the `notify` method is called. We thus characterize the input/output of our `Subject` component by the following set of messages *Msg*.

$$\begin{aligned} Msg = \; & \{\rightarrow sbj.\texttt{Subject}(o_1, o_2), \leftarrow sbj.\texttt{Subject}()\} \\ & \cup \{\rightarrow sbj.\texttt{update}(s), \leftarrow sbj.\texttt{update}()\} \\ & \cup \{\rightarrow o.\texttt{notify}(s), \leftarrow o.\texttt{notify}()\} \end{aligned}$$

The first subset represents the messages dealing with the construction of the `Subject` instance *sbj*. Invocation of the constructor is represented by the message $\rightarrow$ *sbj*.`Subject`($o_1, o_2$) where $o_1$ and $o_2$ are the parameters passed to the invocation. Returning from the constructor is represented by the message $\leftarrow$ *sbj*.`Subject`(). Note that each invocation message has a matching completion message in *Msg*. We assume that the instance variables of the `Subject` class cannot be accessed directly by other objects, i.e., that there is no interaction with a `Subject` object other than through its methods. We can thus solely rely on the previously defined messages to describe the input/output behavior of our component. Using these messages, we give both a state-based and a trace-based specification of the SOP. We assume our specifications to be deterministic in the following way. For each input to the component, there is exactly one output which is specified.

**State-based specification.** A state-based specification is realized using some *abstract* states by describing how a state changes when an event occurs. Our specification of the `Subject` component (Fig. 2) provides an example of a state-based specification in a pre-/postcondition style. In our example, the state consists of a subject, its observers and a stack which stores, in last-in-first-out manner, the states that are available during an update process. The stack is necessary to manage multiple states in the presence of callbacks. We then define the state transitions. A state-based specification consists of a set of specification cases of the following form

**in** *MsgPattern* **out** *MsgPattern* **requires** *pre*; **ensures** *post*;

A transition is specified by a quadruple where the first entry describes the (incoming) message pattern upon which the transition might be triggered. The second entry then describes the outgoing message (i.e., the response) from the component. The third entry forms a precondition over the values of the incoming message pattern and the state upon which the transition might be selected. The fourth entry represents the postcondition and ranges over the values in the pre-state (denoted by *old*(...)), the post-state and the values of the in- and outgoing message. In short, if an incoming message which enters the component fits the incoming message pattern and the precondition *pre* holds, then the component responds by producing an outgoing message and changing its state such that the postcondition *post* holds. To make our specifications short and concise, we assume the part of the state which is not mentioned in the postcondition to retain the same value it had prior to receiving the incoming message.

```
state spec Subject {
  Subject sbj;
  Observer o1, o2;
  Stack<State> st;

  in → sbj.Subject(o1', o2') out ← sbj.Subject()
    requires o1' ≠ o2' ∧ o1' ≠ null ∧ o2' ≠ null;
    ensures o1 = o1' ∧ o2 = o2' ∧ st = Stack.Empty();

  in → sbj.update(s) out → o1.notify(s)
    ensures st = old(st).push(s);

  in ← o.notify() out → o2.notify(s)
    requires o = o1;
    ensures s = old(st).top();

  in ← o.notify() out ← sbj.update()
    requires o = o2;
    ensures st = old(st).pop();
}
```

**Fig. 2.** A state-based specification of the `Subject` class

For the concrete example given above, there are four *cases* which need specifications. Given an instance creation request $\rightarrow$ *sbj*.`Subject`($o_1, o_2$) with the precondition of two distinct and non-null observer parameters, the `Subject` returns the constructor completion message $\leftarrow$ *sbj*.`Subject`() and the new state refers to the two given observers and an empty stack. If the component receives an `update` message, then $o_1$ is notified while the state $s$ is pushed onto the stack. As the control flow leaves the component when the `notify` method is called and control flow enters the component again when the `notify` method returns, we also have to take returning `notify` messages into account. These might happen at two different places; after the first or the second observer has been notified. In our specification, when a notification has finished, we check which observer has been involved in this notification. If it is $o_1$, then the `Subject` proceeds to notify $o_2$ using the state which is on top of the stack *st*. Otherwise, we conclude that the `update` invocation finishes and pop the stack.

A contract-based specification like JML [4] also employs a pre-/postcondition (state-based) specification style. However, as this is based on methods and not message pattern, we can only consider transition descriptions where the incoming message pattern is an invocation and the outgoing message is the corresponding return. We see our specification style as a generalization of the JML pre-/postcondition style as we can additionally handle callbacks.

**Trace-based specification.** Another way to describe the SOP behavior is by using a trace-based specification. Here, the only information used by the spec-

```
trace spec Subject {
  in → sbj.Subject(o'1, o'2) out ← sbj.Subject()
    requires o'1 ≠ o'2 ∧ o'1 ≠ null ∧ o'2 ≠ null;

  in → sbj.update(s) out → o1.notify(s)
    requires sbj = sbj(h);
    ensures o1 = obs1(h);

  in ← o1.notify() out → o2.notify(s)
    requires o1 = obs1(h);
    ensures o2 = obs2(h) ∧ s = getS(h);

  in ← o2.notify() out ← sbj.update()
    requires o2 = obs2(h);
    ensures sbj = sbj(h);
}
```

**Fig. 3.** A trace-based specification of the `Subject` class

ification is the history of messages that have crossed the boundary of the component. This notion of *communication history* [8] is modeled in our specification as a list of incoming and outgoing messages. Similar to before, the specification is described using a pre- and postcondition style, with the difference that we do not use an auxiliary abstract state space but only relate message values to the trace history.[1] Specification cases are of the following form

**in** *MsgPattern* **out** *MsgPattern* **requires** *pre*; **ensures** *post*; ,

where both pre- and postcondition can refer to the history $h$ (also called *history variable* in [9]). The history $h$ contains all messages seen so far, except the ones which were currently matched against the **in** and **out** message pattern.

The specification for the constructor as seen in Fig. 3 is similar to that from the state-based specification. In contrast to the state-based model, upon receiving an `update`, we must go back in the history $h$ to the constructor message to obtain the subject and the first observer that needs to be notified (represented by the extractor functions `sbj` and `obs1` which are not formally stated here).

The trickiest part to specify is when a notification has happened. In the case where the incoming return message comes from $o_1$, we need to notify $o_2$ using the proper state. The problem is that since a callback is allowed and we do not store the states in a stack, we need to obtain this information by emulating the stack effect of each update, which is done by the `getS` function. It searches for the update message containing the state which is relevant to the "stack frame" of the current notification. If the incoming return message comes from $o_2$, the component returns with an `update` completion message.

[1] Note that this is only one possible way to describe the set of admissible traces.

*Comparison.* Both state- and trace-based specification techniques have advantages and disadvantages. On one hand, it is sometimes very difficult in the trace-based approach to extract the necessary information from the trace history (e.g., defining `getS`). The state-based approach allows one to define an abstract state which contains all the relevant information one needs to specify the behavior. On the other hand, state changes must then be described. Some protocol properties are easier to specify in a trace-based way, e.g., using regular expressions.

## 3 Trace-based Model

A trace of a component is a sequence of events which consist of incoming and outgoing messages. It describes how the component responds given a specific instance of an environment which uses the component. A trace-based model can be seen as a collection of such traces which restricts the behavior of the component. We adopt the formalization of messages and traces from [9,10] to define a trace-based model of a sequential deterministic object-based component.

Let *Obj* be a set of objects, *Mtd* a set of methods with unique names, and *Dir* a two-element set $\{\rightarrow, \leftarrow\}$ representing method *invocation* (call) and *completion* (return), respectively. Furthermore, let *Value* be a set of values which encompasses all possible parameter values in actual method calls and return values. Then, with $\overline{v}$ being a shorthand for a possibly empty list of values $v_1, v_2, \ldots$, the set of messages can be defined as follows.

**Definition 1 (Message).** *The set of* messages *Msg (whose instances are denoted by $\mu$) is a subset of $Obj \times Mtd \times List\langle Value\rangle \times Dir$. A tuple $\mu = \langle o, m, \overline{v}, d\rangle$ is a message if the callee $o$ supports the method $m$.*

Instead of representing messages $\mu$ in the tuple format, we depict them graphically: $\rightarrow o.m(\overline{v})$ or $\leftarrow o.m(\overline{v})$. Invocation messages $\rightarrow o.m(\overline{v})$ are grouped into $Msg^{\rightarrow}$ and completion messages into $Msg^{\leftarrow}$, forming a partitioning of *Msg*. We also define extra functions *callee*, *method*, *value*, and *dir* to extract the callee object, method, value and direction elements from a message, respectively. The function *header* extracts the callee and method of a message.

We define a component as a collection of objects $O \subseteq Obj$ (called *component objects*). All other objects (called *environment objects*) are considered as part of the environment $O_{env} = Obj \backslash O$.

Based on this partitioning, we can categorize the messages into incoming and outgoing messages from the perspective of the component. An *incoming message* is either an invocation message whose callee is a component object, or a completion message whose callee is an environment object. An *outgoing message* is either an invocation message whose callee is an environment object, or a completion message whose callee is a component object. As we are interested in modeling the observable behavior of the component, we only deal with traces composed of alternating incoming and outgoing messages.

**Definition 2 (Component Trace).** *Given a set of messages Msg and a component $O$, a* component trace *$t$ is a (possibly empty or infinite) sequence of pairs*

*of incoming and outgoing messages* $(\mu_1, \mu_2), (\mu_3, \mu_4), \ldots$*, where odd-indexed messages are incoming and even-indexed messages outgoing.*[2]

Note that in the definition above, we assume that the component may not diverge. To include divergence, we could extend *Msg* to include a special message symbol (e.g., $\perp$) to indicate this situation.

In the sequential setting, a component trace must follow the call stack property, where method completions must appear in reverse order of the corresponding method invocations. This is captured by the following definition of well-formed component traces.

**Definition 3 (Well-formed Component Trace).** *A non-empty component trace* $t = \mu_1, \mu_2, \ldots$ *is* well-formed *with respect to a component* $O$ *and the environment* $O_{env} = Obj \backslash O$*, iff for each completion message there exists a prior invocation message such that the call stack property (predicate match) holds:* $\forall \mu_j \in Msg^{\leftarrow} \bullet \exists k < j \bullet match(k, j)$*, where*

$$\begin{aligned} match(a,b) &\stackrel{def}{=} \mu_a \in Msg^{\rightarrow} \wedge \mu_b \in Msg^{\leftarrow} \wedge \\ &\quad header(\mu_a) = header(\mu_b) \wedge split(a+1, b-1), \text{ and} \\ split(a,b) &\stackrel{def}{=} a > b \vee match(a,b) \vee \\ &\quad \exists\ a < c < b-1 \bullet split(a,c) \wedge split(c+1,b) \end{aligned}$$

*An empty trace is well-formed.*

The well-formedness condition above states that every completion message needs to have a matching invocation message. The proper matching has to be selected such that no two completion messages are matched to a single invocation message, which is captured by the *match* predicate. The *match* and *split* predicates are mutually recursive, where *match* matches the invocation and completion messages at the two ends of the subtrace (i.e., $\mu_a$ and $\mu_b$), and *split* partitions the rest of the subtrace such that for each partition, we can form the matching. Other sequential properties[3] are not described as they complicate the definitions and proofs without adding substantial insight.

We define $Traces(O)$ as the set of well-formed traces of the component $O$. As we do not specify the behavior of the environment, it may decide to terminate at any moment. Therefore, we require the set $Traces(O)$ to be prefix-closed, i.e., for each trace $t \in Traces(O)$, all its prefixes must also be in $Traces(O)$. A trace-based model is then simply defined using the set of traces.

**Definition 4 (Trace-based Model).** *Given a set of messages Msg and a component* $O$*, a* trace-based model *is the set of well-formed traces:*

$\mathcal{T}(Msg, O) = Traces(O)$.

[2] We drop the brackets surrounding the pairs whenever it is clear from the context.

[3] Examples of other sequential properties: all traces begin with the construction of a component; a component cannot send a message to an object from the environment before that object has been introduced to the component, etc.

Because in this paper the set of messages *Msg* and objects of the component $O$ is clear from the context, we simply write $\mathcal{T}$ and *Traces* for the trace-based model and its set of well-formed traces, respectively.

Since we want a model for a deterministic component, we also require that the component acts as a function given some trace prefix and incoming message to produce exactly one outgoing message. Stated formally,

$$\forall t, t', t'' \in \mathit{Traces} \bullet t' = t, \mu_{2n-1}, \mu_{2n} \wedge t'' = t, \mu'_{2n-1}, \mu'_{2n} \wedge \mu_{2n-1} = \mu'_{2n-1} \implies \mu_{2n} = \mu'_{2n} \ .$$

*Example (Trace-based model for the* `Subject` *component).* The trace-based model $\mathcal{T}_{subj}$ can be defined using the set of well-formed traces $\mathit{Traces}_{subj}$ generated by the specification in Fig. 3. The set of traces $\mathit{Traces}_{subj}$ is inductively constructed. The empty trace is element of the set. Then, for any element $t$ of the set, the trace $t' = t, \mu_1, \mu_2$ is also element of the set, if it satisfies the following conditions. There must be a specification case with **in** and **out** message pattern that are matched by $\mu_1$ and $\mu_2$ and where both pre- and postconditions are satisfied. Furthermore, the trace $t'$ must be well-formed (see Def. 3).

## 4 State-based Model

Another way to specify the behavior of a component is to determine how the component would act given a request from the environment by looking at the state of the component, updating the state, and forming a response based on it. This is captured by the well-known notion of transition system. In this section, we describe state-based models, defined as transition systems, and how we can represent the trace-based models as state-based models.

**Definition 5 (State-based Model).** *Given a set of messages Msg and a component $O$, a* state-based model $\mathcal{M}(\mathit{Msg}, O)$ *is a triple $\langle S, \Theta, s_0 \rangle$ where $S$ is a set of states, $s_0 \in S$ is the initial state and $\Theta \subseteq S \times \mathit{Msg} \times \mathit{Msg} \times S$ is the transition relation, where the first message represents an incoming message and the second one an outgoing message.*

As with trace-based models, we drop the set of messages *Msg* and objects of the component $O$ from the argument of $\mathcal{M}$. We also represent a transition between two states $s, s'$ graphically as $s \xrightarrow{\mu_a, \mu_b} s'$. A (finite) component trace $t = (\mu_1, \mu_2), \ldots$ is induced by $\mathcal{M}$ if there exists a sequence of states $s_0, s_1, \ldots$ such that $\forall i > 0 \bullet s_{i-1} \xrightarrow{\mu_{2i-1}, \mu_{2i}} s_i$.

We require our state-based models to be *concise*. A model is concise if each state is reachable by some sequence of transitions from the initial state.

*Example (State-based model of the* `Subject` *component).* The state-based model $\mathcal{M}_{subj} = \langle S, \Theta, s_0 \rangle$ is a state-based model of the `Subject` component specified in Fig. 2. It has $S \subseteq$ `Subject` $\times$ `Observer` $\times$ `Observer` $\times$ `Stack<State>` as set of states and $s_0 =$ (**`null,null,null,null`**) as initial state. The transition

relation $\Theta$ can be derived from the state-based specification in a similar way as how the trace-based model was derived from the trace-based specification. Note however that no well-formedness property must yet hold.

The lemma below formulates trace-based models in terms of state-based models.

**Lemma 1.** *Every trace-based model $\mathcal{T}$ can be canonically represented as a state-based model $\mathcal{M} = \langle S, \Theta, s_0 \rangle$.*

*Proof.* By construction. Let $S$ be a set of finite traces from *Traces* and $s_0$ be the empty trace $\epsilon$. The transition relation $\Theta$ can be built in the following way. We start from an empty relation. Now take any two elements $t, t'$ of *Traces* such that $t' = t, \mu_a, \mu_b$. Then, $t \xrightarrow{\mu_a, \mu_b} t'$ is added to $\Theta$. This construction is similar to how one would build a trie (re*trie*val tree [11]) from a set of strings. See Appendix for more details.

The usual notion of determinism for transition systems is that for any given state $s$ and a transition label $(\mu_a, \mu_b)$ there is at most one state $s'$ such that $s \xrightarrow{\mu_a, \mu_b} s'$. To include the desired notion of determinism stated in the previous section, we need to strengthen this notion by also requiring that for any state $s$ and an incoming message $\mu_a$, there is at most one outgoing message $\mu_b$ and one next state $s'$ such that $s \xrightarrow{\mu_a, \mu_b} s'$. It is not enough to use the determinism of the trace-based view, since the resulting state-based model may be non-deterministic in traditional sense while inducing the same set of traces.

Definition 5 alone is not strong enough to ensure that the resulting component traces are well-formed. There are, for example, state-based models which may induce a trace which breaks the call stack requirement. To enforce the well-formedness requirement, one can build a specific model for each requirement (called *restrictor models*), and then take the synchronous product between the restrictor models and the original state-based model.

**Definition 6 (Synchronous Product).** *Given two state-based models $\mathcal{M}_i = \langle S_i, \Theta_i, s_{0,i} \rangle, i = 1, 2$, their* synchronous product *$\mathcal{M}' = \mathcal{M}_1 \otimes \mathcal{M}_2$ is $\langle S', \Theta', s'_0 \rangle$ where $S' \subseteq S_1 \times S_2$, $s'_0 = (s_{0,1}, s_{0,2})$, and $\Theta' \subseteq S' \times Msg \times Msg \times S'$ is the least relation such that $(s_1, s_2) \xrightarrow{\mu_a, \mu_b}{}' (s'_1, s'_2)$ if $s_1 \xrightarrow{\mu_a, \mu_b}_1 s'_1$ and $s_2 \xrightarrow{\mu_a, \mu_b}_2 s'_2$.*

As the only requirement to have a well-formed state-based model is the call stack property, it is enough to build a restrictor model which induces traces following the call stack property.

**Definition 7 (Call Stack Restrictor Model).** *Let $Header \subseteq Obj \times Mtd$ be the set of all message headers. The* call stack restrictor model *$\mathcal{C} = \langle S_c, \Theta_c, s_{0,c} \rangle$ is a state-based model whose state is a stack of elements of Header, with the initial state being the empty stack[4]. The transition $s_1 \xrightarrow{\mu_a, \mu_b}_c s_2$ exists if one of the following conditions hold.*

[4] We denote the empty stack as $\epsilon$ and a non-empty stack as $h' = h : hs$, where $h$ is the top element and $hs$ denotes the rest of the stack.

1. $dir(\mu_a) = \rightarrow \wedge\ dir(\mu_b) = \rightarrow \wedge\ s_1 = hs \wedge s_2 = header(\mu_b) : header(\mu_a) : hs$
2. $dir(\mu_a) = \leftarrow \wedge\ dir(\mu_b) = \leftarrow \wedge\ s_1 = header(\mu_a) : header(\mu_b) : hs \wedge s_2 = hs$
3. $dir(\mu_a) = \rightarrow \wedge\ dir(\mu_b) = \leftarrow \wedge\ header(\mu_a) = header(\mu_b) \wedge s_1 = s_2$
4. $dir(\mu_a) = \leftarrow \wedge\ dir(\mu_b) = \rightarrow \wedge\ s_1 = header(\mu_a) : hs \wedge s_2 = header(\mu_b) : hs$

Note that our restrictor model is non-deterministic. However, since our state-based model is deterministic, the product will remain deterministic.

**Definition 8 (Well-formed State-based Model).** *Given a state-based model $\mathcal{M}$, the* well-formed state-based *model for $\mathcal{M}$ is $\mathcal{M}_{wf} = \mathcal{M} \otimes \mathcal{C}$.*

*Example (Well-formed state-based model of the* `Subject` *component).* The well-formed state-based model of the example is $\mathcal{M}_{wfsubj} = \mathcal{M}_{wfsubj} \otimes \mathcal{C}_{subj}$. In order to ensure the call stack property, the states of $\mathcal{M}_{wfsubj}$ are thus a subset of `Subject` × `Observer` × `Observer` × `Stack<State>` × `Stack<Header>`.

## 5 Model Abstraction

We are interested in reducing the size of our models without altering the component trace sets represented by the models. To do this, the models need to be related with each other. One such relation is the state abstraction relation (cf. §7.4 of [12]), which is an instance of the simulation relation [13]. We extend this relation to our setting and reuse known results to build a most abstract model.

**Definition 9 (State Abstraction).** *Given two state-based component models $\mathcal{M}_i = \langle S_i, \Theta_i, s_{0,i} \rangle$, $i = 1, 2$, we say that $\mathcal{M}_2$ is* more abstract *than $\mathcal{M}_1$ iff there is a total abstraction function $\alpha : S_1 \rightarrow S_2$ such that*
$s_{0,2} = \alpha(s_{0,1})$*; and if* $s_1 \xrightarrow{\mu_a, \mu_b}_1 s_1'$*, then* $\alpha(s_1) \xrightarrow{\mu_a, \mu_b}_2 \alpha(s_1')$.

*Example (Abstraction of the trace-based model of the* `Subject` *component).* Using Lemma 1, we build $\mathcal{M}_{tsubj}$ as the state-based model from the trace-based model $\mathcal{T}_{subj}$. We compare $\mathcal{M}_{tsubj}$ and $\mathcal{M}_{wfsubj}$ by providing an abstraction function $\alpha : S_{tsubj} \rightarrow S_{wfsubj}$ as follows.

$\alpha(\epsilon) =$ (**null**, **null**, **null**, **null**, $\epsilon$)

$\alpha((\rightarrow sbj.\texttt{Subject}(o_1, o_2), \leftarrow sbj.\texttt{Subject}()) : \epsilon) =$
(*sbj*, $o_1$, $o_2$, $\epsilon$, $\epsilon$)

$\alpha((\rightarrow sbj.\texttt{update}(s), \rightarrow o.\texttt{notify}(s)) : l) =$
(*sbj*, $o_1$, $o_2$, $s : sl$, $o.\texttt{notify} : sbj.\texttt{update} : hl$)
**where** $(sbj, o_1, o_2, sl, hl) = \alpha(l),\ o = o_1$

$\alpha((\leftarrow o.\texttt{notify}(), \rightarrow p.\texttt{notify}(st)) : l) =$ $(sbj, o_1, o_2, s : sl, p.\texttt{notify} : hl)$
**where** $(sbj, o_1, o_2, s : sl, o.\texttt{notify} : hl) = \alpha(l),\ o = o_1,\ p = o_2,\ st = s$

$\alpha((\leftarrow o.\texttt{notify}(), \leftarrow sbj.\texttt{update}()) : l) =$ $(sbj, o_1, o_2, sl, hl)$
**where** $(sbj, o_1, o_2, s : sl, o.\texttt{notify} : sbj.\texttt{update} : hl) = \alpha(l),\ o = o_2$

For the initial state, we map the empty history to **null**s and the empty header stack. Each of the remaining cases shows a one-to-one correspondence between the postconditions defined in the trace-based specification (Fig. 3) and the postconditions defined in the state-based specification (Fig. 2).

Because such an abstraction function $\alpha$ exists, we can conclude that the trace-based model of the `Subject` component is more abstract than the state-based model of the `Subject` component derived from the state-based specification. The following main theorem states this result in a more general way.

**Theorem 1.** *For any trace-based model $\mathcal{T}$, there is a well-formed state-based model $\mathcal{M}_{wf}$ which is more abstract than $\mathcal{T}$. The converse does not hold.*

*Proof.* The first part of this lemma follows directly from Lemma 1 and taking the identity function as $\alpha$. For the converse, consider a specification which concentrates only on the well-formedness property of the traces. Any well-formed trace in the trace-based model of the specfication which has completed all updates can be abstracted to the state with empty stacks in the state-based model. As the abstraction must be a function, the converse does not hold.

We can go further than the main theorem by adapting well-known results about state transition systems. We first define the standard notion of simulation.

**Definition 10 (Simulation [13]).** *Let $\mathcal{M}_i = \langle S_i, \Theta_i, s_{0,i} \rangle$, $i = 1, 2$ be state-based models over Msg. A* simulation *for $(\mathcal{M}_1, \mathcal{M}_2)$ is a binary relation $\mathcal{R} \subseteq S_1 \times S_2$ such that $(s_{0,1}, s_{0,2}) \in \mathcal{R}$; and if $(s_1, s_2) \in \mathcal{R}$ and $s_1 \xrightarrow{\mu_a, \mu_b}_1 s_1'$, then there is $s_2' \in S_2$ such that $s_2 \xrightarrow{\mu_a, \mu_b}_2 s_2'$ and $(s_1', s_2') \in \mathcal{R}$.*

If there exists an simulation relation $\mathcal{R}$ for $(\mathcal{M}_1, \mathcal{M}_2)$, we say that $\mathcal{M}_1$ is *simulated* by $\mathcal{M}_2$, denoted $\mathcal{M}_1 \preceq \mathcal{M}_2$. Furthermore, if the simulation relation occurs in both directions (i.e., $\mathcal{M}_1 \preceq \mathcal{M}_2$ and $\mathcal{M}_2 \preceq \mathcal{M}_1$), we say that $\mathcal{M}_1$ is *simulation equivalent* to $\mathcal{M}_2$, denoted by $\mathcal{M}_1 \simeq \mathcal{M}_2$. We are only interested in the (unique [14]) *maximal* simulation relation, which is a simulation relation that contains all other simulation relations between the two models.

The following lemma shows that the abstraction is a simulation.

**Lemma 2.** *If $\mathcal{M}_2$ is more abstract than $\mathcal{M}_1$, then $\mathcal{M}_1 \preceq \mathcal{M}_2$.*

*Proof.* Consider $\mathcal{M}_i = \langle S_i, \Theta_i, s_{0,i} \rangle$, $i = 1, 2$ and $\alpha : S_1 \to S_2$ to be the abstraction function. We simply take $\mathcal{R} = \{(s, \alpha(s)) \mid s \in S_1\}$.

A most abstract model is a model which up to renaming has no more abstraction. To construct it, we need the notion of equivalence classes.

**Definition 11 (Equivalence Classes).** *Let $S$ be a set and $\mathcal{R}$ an equivalence on $S$. For $s \in S, [s]_{\mathcal{R}} = \{s' \mid (s, s') \in \mathcal{R}\}$. The quotient space of $S$ under $\mathcal{R}$ is defined as $S/\mathcal{R} = \{[s]_{\mathcal{R}} \mid s \in S\}$.*

By building the equivalence classes of the states based on the simulation relation, we end up with the simulation quotient model.

**Definition 12 (Simulation Quotient Model).** *Given a state-based model $\mathcal{M} = \langle S, \Theta, s_0 \rangle$, the* simulation quotient state-based model *$\mathcal{M}/\simeq$ is a triple $\langle S/\simeq, \Theta_{\simeq}, [s]_{\simeq} \rangle$, where $[s] \xrightarrow{\mu_a, \mu_b}_{\simeq} [s']$ if $s \xrightarrow{\mu_a, \mu_b} s'$.*

Grumberg and Bustan [15] show that the simulation quotient model $\mathcal{M}/\simeq$ is the unique smallest (in terms of number of states) state-based model up to renaming which is simulation equivalent to $\mathcal{M}$. Since the quotient space is a partitioning of the original state set $S$, we can also see it as an abstraction.

**Theorem 2 (Most Abstract Model [15]).** *The simulation quotient model $\mathcal{M}/\simeq$ is the most abstract model of the state-based model $\mathcal{M}$ up to renaming.*

*Example (Simulation quotient model of the* `Subject` *component).* The following lemma states that the well-formed state-based model of the `Subject` component specification is a simulation quotient model for the specification.

**Lemma 3.** *$\mathcal{M}_{wfsubj}$ is the simulation quotient model for $\mathcal{M}_{tsubj}$.*

*Proof.* By contradiction. Assume $\mathcal{M}_{wfsubj}$ is not the simulation quotient model. Then there exist two states of $\mathcal{M}_{wfsubj}$ that are equivalent, which is false. See Appendix for complete sketch.

By Theorem 2, $\mathcal{M}_{wfsubj}$ is the unique smallest state-based model for the `Subject` component specifications and, hence, its most abstract model.

In the deterministic setting, the notions of simulation equivalence and trace equivalence collapse [16]. As a result, the quotient model is a most abstract model, i.e., a model such that there is no other smaller state-based model that is more abstract and still retains exactly the same behavior.

In a non-deterministic setting, the abstraction function defined in Def. 9 only guarantees finite trace inclusion, while the simulation equivalence guarantees finite trace equivalence [17]. In general if we want a more abstract model to remain behaviorally equivalent to the abstracted model, then we need to use bisimulation equivalence [18] as the reduction. This requires a more general definition of abstraction, where, in addition to the definition of abstraction given here, the abstract states need to be related back to the abstracted states.

## 6 Related Work

State-based models are usually, in the object-oriented setting, specified using contracts as introduced in Eiffel [3]. JML [4] and Spec# [19] adopt this approach to allow modular specifications in Java and C#, respectively. Contracts mainly consist of method pre-/postconditions and class invariants, making it non-trivial to deal directly with callbacks as in the subject/observer example (Sect. 2).

A generalized state-based model similar to the one given here is present in the $Z^{++}$ methodology [20], where object-oriented real-time systems are specified using real-time logic. While similar notions of message predicates are used within the state-based specifications, their purpose is for timing measurement.

Trace-based specifications are well-known in the literature of processes and modules [21,22]. This trace idea has also appeared in the object-oriented setting as early as [8] (communication histories), further developed in [23] and later in [24,25], especially for modeling of open asynchronous distributed systems.

In connection to proof systems, trace-based specifications have been used most recently in [9,26] in the Creol [27] setting. The trace of a component is seen as a projection from the global trace and checked against invariants which capture communication patterns in form of regular expressions of messages.

At the program level, the idea to characterize the behavior of object-oriented components by admissible traces is also used in the context of observational equivalence to give a fully abstract semantics for object-oriented languages [5,28].

Jass [29] gives specifications in the form of trace assertions, whose semantics is based on CSP. This technique allows to specify our subject example in a similar fashion. However, a clear component model is missing. Similarly, Cheon and Perumandla [30] extend JML to introduce assertions based on call sequences in form of regular expressions, by abstracting from argument values and callee.

## 7 Conclusion and Future Work

In this paper we have shown for a sequential object-based setting that state-based models are abstractions of trace-based models. We have also given an example of a subject component illustrating the relation between state- and trace-based specification approaches by describing them in a common framework.

As the subtitle of this paper suggests, our long-term goal is to formalize this generalization to describe precisely the behavior of a component in an object-oriented setting. We are especially interested in defining the connection between such mixed trace- and state-based specifications and programming languages. An ideal connection would allow specification composition and modular verification.

Another natural extension to this work is to consider a more general setting such as full object-orientation (i.e., allowing subtyping and inheritance) and concurrency. In addition, sometimes we would like to specify how the component should be used. For example, we want to guarantee that the observer, upon being notified, actually responds to the subject component (i.e., does not diverge). By specifying such restrictions about the environment, the set of traces is no longer prefix-closed, and thus the semantics of our state-based model has to be altered.

**Acknowledgements.** The authors would like to thank the anonymous referees for their constructive comments and suggestions.

## References

1. Müller, P.: Modular Specification and Verification of Object-Oriented Programs. PhD thesis, FernUniversität Hagen (2001)
2. Cheon, Y., Leavens, G., Sitaraman, M., Edwards, S.: Model variables: Cleanly supporting abstraction in design by contract. Softw. Pract. Exper. **35**(6) (2005) 583–599
3. Meyer, B.: Object-Oriented Software Construction. Prentice-Hall (1988)
4. Leavens, G.T., Baker, A.L., Ruby, C.: Preliminary design of JML: A behavioral interface specification language for Java. ACM SIGSOFT Software Engineering Notes **31**(3) (2006) 1–38

5. Jeffrey, A., Rathke, J.: Java Jr: Fully abstract trace semantics for a core Java languages. In: ESOP. Volume 3444 of LNCS., Springer (2005) 423–438
6. Kyas, M.: Verifying OCL Specifications of UML Models: Tool Support and Compositionality. PhD thesis, Leiden University (2006)
7. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns. Addison-Wesley, Boston, MA (January 1995)
8. Dahl, O.J.: Can program proving be made practical? In: Les Fondements de la Programmation. (1977) 57–114
9. Dovland, J., Johnsen, E.B., Owe, O.: Observable behavior of dynamic systems: Component reasoning for concurrent objects. ENTCS **203**(3) (2008) 19–34
10. Kyas, M., de Boer, F.S., de Roever, W.P.: A compositional trace logic for behavioural interface specifications. NJC **12**(2) (2005) 116–132
11. Fredkin, E.: Trie memory. CACM **3**(9) (1960) 490–499
12. Baier, C., Katoen, J.P.: Principles of Model Checking (Representation and Mind Series). The MIT Press (2008)
13. Milner, R.: An algebraic definition of simulation between programs. In: IJCAI. (1971) 481–489
14. Grumberg, O., Long, D.E.: Model checking and modular verification. ACM TOPLAS **16**(3) (1994) 843–871
15. Bustan, D., Grumberg, O.: Simulation-based minimazation. ACM TOCL **4**(2) (2003) 181–206
16. Kucera, A., Mayr, R.: Simulation preorder over simple process algebras. Information and Computation **173**(2) (2002) 184–198
17. van Glabbeek, R.J.: The linear time-branching time spectrum (extended abstract). In: CONCUR. Volume 458 of LNCS., Springer (1990) 278–297
18. Park, D.: Concurrency and automata on infinite sequences. In: Proceedings of the 5th GI-Conference on TCS, London, UK, Springer-Verlag (1981) 167–183
19. Barnett, M., Leino, K.R.M., Rustan, K., Leino, M., Schulte, W.: The Spec# programming system: An overview. In: CASSIS, Springer (2004) 49–69
20. Lano, K.: Formal Object-Oriented Development. Springer-Verlag New York, Inc., Secaucus, NJ, USA (1995)
21. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall (1985)
22. Bartussek, W., Parnas, D.L.: Using assertions about traces to write abstract specifications for software modules. In: ECI. Volume 65 of LNCS., Springer (1978) 211–236
23. Nierstrasz, O.: Regular types for active objects. In: OOPSLA. (1993) 1–15
24. Johnsen, E.B., Owe, O.: A compositional formalism for object viewpoints. In: FMOODS. Volume 209 of IFIP Conference Proceedings., Kluwer (2002) 45–60
25. Kyas, M., de Boer, F.S.: On message specification in OCL. In: Compositional Verification in UML. Volume 101 of ENTCS., Elsevier (2004) 73–93
26. Ahrendt, W., Dylla, M.: A verification system for distributed objects with asynchronous method calls. In: ICFEM. Volume 5885 of LNCS., Springer (2009) 387–406
27. Johnsen, E.B., Owe, O.: An asynchronous communication model for distributed concurrent objects. Software and System Modeling **6**(1) (2007) 39–58
28. Steffen, M.: Object-connectivity and observability for class-based, object-oriented languages. (2006) Habilitation Thesis, University of Kiel.
29. Bartetzko, D., Fischer, C., Möller, M., Wehrheim, H.: Jass - Java with assertions. ENTCS **55**(2) (2001) 1–15
30. Cheon, Y., Perumandla, A.: Specifying and checking method call sequences of Java programs. Software Quality Journal **15**(1) (2007) 7–25

## Appendix: Proof for Lemmas 1 and 3 and Theorem 1

**Lemma 1.** Every trace-based model $\mathcal{T}$ can be canonically represented as a state-based model $\mathcal{M} = \langle S, \Theta, s_0 \rangle$.

*Proof.* Let us construct $\mathcal{M} = \langle S, \Theta, s_0 \rangle$ from $\mathcal{T} = \mathit{Traces}(O)$.

Let $S \subseteq (\mathit{Msg} \times \mathit{Msg})^*$. The first part of a state is a finite prefix of a trace, and the second part indicates whether the prefix stored in that state is indeed a finite trace contained in $\mathit{Traces}(O)$. We let $s_0 = \epsilon$.

Initially, $S = \{s_0\}$ and $\Theta$ is an empty relation. We populate them inductively as follows. For each two consecutive finite traces $t = \mu_{i,1}, \mu_{o,1}, \ldots, \mu_{i,n}, \mu_{o,n}$ and $t' = t, \mu_{i,n+1}, \mu_{o,n+1}$, such that $t, t' \in \mathit{Traces}(O)$ and $t$ ends in $s \in S$, we add $s' = (t')$ to $S$ and $s \xrightarrow{\mu_{i,n+1}, \mu_{o,n+1}} s'$ to $\Theta$.

The construction given in this proof is similar to the construction of a trie (from re*trie*val tree [11]): a tree data structure which stores a dictionary, where each node represents a common prefix of at least a string present in the dictionary. This construction ensures that every state is reachable from the initial state. Thus, the resulting state-based model is concise and since $\mathit{Traces}(\mathcal{M})$ is prefix-closed, $\mathit{Traces}(\mathcal{M}) = \mathit{Traces}(O)$.

The construction also ensures that each well-formed trace is induced by a unique path on the state-based model. Therefore, the product between this model and the call stack restrictor model will yield the same model. Hence, this construction produces a well-formed model. □

**Lemma 3.** $\mathcal{M}_{wfsubj}$ is the simulation quotient model for $\mathcal{M}_{tsubj}$.

*Proof.* By contradiction. Assume that there exists such a more abstract model $\mathcal{M}'$ which has less state than $\mathcal{M}$. This means that there exist two states in $\mathcal{M}$ that is represented by a single state in $\mathcal{M}'$.

It is trivial to see that the two observers have to remain present in the state so that the subject knows which two observers (and also the order) it has to notify after receiving an update.

Assume now that we have two states whose state stack contents are $s_1, s_2, \ldots, s_k, \ldots, s_n$ and $s_1, s_2, \ldots, s'_k, \ldots, s_n$ which is merged together in $\mathcal{M}'$. Therefore, we have (at least) two possible executions to reach this state, namely $n$ consecutive `update` callbacks, one with the $k^{\text{th}}$ being `update` $(s_k)$, the other with `update` $(s'_k)$. Playing the environment, we now stop the `update`s and allow the pending notifications for the second observer to be completed without any callbacks. However, at the time we need to send the notification about $s_k$ or $s'_k$, we have no way to distinguish them in $\mathcal{M}'$. Therefore, these two states cannot be merged together. Since we choose arbitrary $k$ to do this, there exists no two states in $\mathcal{M}$ that can be merged together.

A similar argument can be applied for the case of two states whose header stacks are different. Hence, we achieve our contradiction. □

**Theorem 3.** For any trace-based model $\mathcal{T}$, there is a well-formed state-based model $\mathcal{M}_{wf}$ which is more abstract than $\mathcal{T}$. The converse does not hold.

*Proof.* The first part of this lemma follows directly from Lemma 1 and taking the identity function as $\alpha$. For the converse, consider a specification which concentrates only on the well-formedness property of the traces. Any well-formed trace in the trace-based model of the specfication which has completed all updates can be abstracted to the state with empty stacks in the state-based model. As the abstraction must be a function, the converse does not hold.