


# Automatic Cross Validation of Multiple Specifications: A Case Study\*

View metadata, citation and similar papers at [core.ac.uk](http://core.ac.uk)

brought to you by  CORE

provided by Archivio istituzionale della ricerca - Politecnico di Milano

Politecnico di Milano

DeepSE Group, Dipartimento di Elettronica e Informazione  
p.za Leonardo da Vinci 32, 20133 Milano (MI) Italy  
{ghezzi,mocci,salvaneschi}@elet.polimi.it

**Abstract.** The problem of formal software specification has been addressed and discussed since the infancy of software engineering. However, among all the proposed solutions, none is universally accepted yet. Many different formal descriptions can in fact be given for the same software component; thus, the problem of determining the consistency relation among those descriptions becomes relevant and potentially critical. In this work, we propose a method for comparing two specific kinds of formal specifications of containers. In particular, we check the consistency of intensional behavior models with algebraic specifications. The consistency check is performed by generating a behavioral equivalence model from the intensional model, converting the algebraic axioms into temporal logic formulae, and then checking them against the model by using the NuSMV model checker. An automated software tool which encodes the problem as model checking has been implemented to check the consistency of recovered specifications of relevant Java classes.

## 1 Introduction and Motivations

Given a software component, its *specification* is a description of its functionality, guaranteed by its provider, upon which clients can rely [1]. Although the problem of formally and precisely specifying software has been discussed through all the history of software engineering, none of the proposed solutions has been universally accepted yet. For almost every specification methodology, it is possible to distinguish between a *syntactic part*, which describes the component's signature, and a *semantic part*, which describes the behavior of the component in terms of visible effects for the clients. The difficult problems are in the semantic part.

Different descriptions can in fact be given for the same software component. A possible classification of specifications distinguishes between *operational* and *descriptive* specifications [2]. An operational specification describes the desired behavior by providing a model implementation of the system, for example by using abstract automata. Examples of operational specifications are state machine models (e.g. [3]). Another different specification style is through descriptive

---

\* This research has been partially funded by the European Commission, Programme IDEAS-ERC, Project 227977-SMScom.

specifications, which are used to state the desired properties of a software component by using a declarative language, for example by using logic formulae. Examples of such notations are *JML* [4] or *algebraic specifications* [5]. Different specification styles (and languages) may differ in their expressiveness and very often their use depends on the preference and taste of the specifier, the availability of support tools, etc. Moreover, sometimes different specifications for the same piece of software are provided to describe different viewpoints. Living with multiple specifications of the same entity, however, can be dangerous. In particular, a question naturally arises about their *consistency* or even their *equivalence*. Intuitively, let us consider two specifications  $A$  and  $B$ . We say that  $A$  is consistent with  $B$  if all the behaviors specified by  $A$  are also specified by  $B$ . The equivalence problem can be stated as mutual consistency, that is, we consider  $A$  and  $B$  equivalent if and only if  $A$  is consistent with  $B$  and vice-versa. In general, it is not possible to formally state a precise definition of consistency without instantiating the specific formalisms used to express  $A$  and  $B$ . Another relevant problem of software specification is that its production might be as expensive as coding. This difficulty is why specifications are often partial, given informally, or they are completely absent. To address this issue, recent research [6, 7, 8] has proposed several techniques for automatic recovery.

This paper casts the general problem of automatically comparing two formal specifications of stateful components into two instance specification languages. It proposes an automated methodology to check *algebraic specifications* against *intensional behavior models* [9] by using symbolic model checking [10]. For both specification techniques, inference methods and tools are available: algebraic specifications can be recovered with a tool named HEUREKA [8] and intensional behavior models can be inferred by our recent work SPY [7]. However, the possibility to extract the specifications is not essential to the proposed approach; it will be used only to leverage an empirical evaluation of the contribution based on recovered specifications. Even if the specification comparison methodology is not restricted to any particular kind of software components, the specifications styles are particularly suitable for classes implementing containers. For this reason, we will consider containers as the case study entities for which we apply our specification consistency method. Containers are rather complex abstract data types, implemented by components with infinite states. For instance, consider a set of strings. Let strings be defined over a finite alphabet  $I$ ; their cardinality is  $|I^*| = |\mathbb{N}|$ . If we now consider containers implementing sets of strings in  $I^*$ , their cardinality is  $|\wp(\mathbb{N})|$ . Thus, when dealing with containers, we are possibly dealing with components with a number of states which may be non-denumerable. To avoid intrinsic unmanageable complexity, in this paper we address the problem of specification consistency with a specific limitation. We do not aim at finding a proof of consistency of two specifications, which may require complex formalisms and would hardly be automated. Instead, we cast the problem by providing an automatic way of comparing the behavioral information prescribed by the specifications under a finite subset of the behaviors of the component, and we guarantee that under that limit the specifications are either consistent

or not. Intuitively, the proposed approach instantiates an intensional behavior model as a finite state machine (a *BEM*, *Behavioral Equivalence Model*), whose states represent behaviorally equivalent classes of component instances, and algebraic specifications are finitized and translated into temporal logic formulae. Algebraic specifications play the role of properties to be verified against a limited and partial model of the component. This approach has been implemented and extensively tested; in particular, we verified the consistency of specifications recovered from relevant number of classes coming from the JAVA library.

A justification for the analysis of such complex specifications by instantiating them to finite models can be found in the so-called *small scope hypothesis* [11]. This hypothesis is fundamental when dealing with large state spaces; intuitively, it states that *most bugs have small counterexamples*. Within our context, the hypothesis can be formulated as follows: if the specifications are not consistent, a counterexample which shows the inconsistency is likely to be found in small and partial models of the specifications. Conversely, if the analysis does not show any counterexample, in theory we cannot conclude anything about their consistency, but in practice it is *very unlikely* that the two specifications are inconsistent.

This paper is organized as follows. Section 2 illustrates algebraic specifications and intensional behavior models and details the problem of comparing those two specification techniques. Section 3 describes the proposed approach for checking algebraic specifications against intensional behavior models. Section 4 provides empirical evaluation of the methodology. Section 5 discusses related work in the state of the art. Finally, Section 6 outlines conclusions and future work.

## 2 Specifying Containers

This section illustrates two different techniques, intensional behavior models and algebraic specifications, that can be used to specify the behavior of stateful components. Both techniques focus on software components implementing containers. Such components are designed according to the principle of *information hiding*, that is, clients cannot access the data structures internal to the component, but they must interact with it by using a set of methods which define the *interface* of the component. To illustrate the two specification techniques, we refer to the *Deque* container, which is inspired by the *ArrayDeque* class of the JAVA library. Essentially, the class is a double-ended queue, that is, a queue that also supports LIFO removal through the *pop* operation. Figure 1 illustrates the interface of this data abstraction when strings are used as contained objects.

```
public class Deque {
    public Deque() { .. } public void push(String elem) { .. }
    public String pop() { .. } public String deq() { .. }
    public Integer size() { .. }
}
```

**Fig. 1.** The public interface of *Deque*

**sorts :**  $Deque, String, Boolean, Integer$   
**operations :**  
 $deque : \rightarrow Deque$   
 $push : Deque \times String \rightarrow Deque; \quad size : Deque \rightarrow Integer$   
 $deque.state : Deque \rightarrow Deque; \quad deque.retval : Deque \rightarrow String \cup \{Exception\}$   
 $pop.state : Deque \rightarrow Deque; \quad pop.retval : Deque \rightarrow String \cup \{Exception\}$   
**axioms :**  $\forall x \in Deque \ e, f \in String$   
 $pop.state(push(x, e)) = x; \quad pop.retval(push(x, e)) = e$   
 $pop.state(Deque()) = Deque(); \quad pop.retval(Deque(), e) \rightsquigarrow Exception$   
 $deque.state(push(push(x, e), f)) = push(deq(push(x, e), f))$   
 $deque.state(push(Deque(), e)) = Deque()$   
 $deque.state(Deque()) = Deque(); \quad deque.retval(push(Deque(), e)) = e$   
 $deque.retval(push(push(x, e), f)) = deque.retval(push(x, e))$   
 $deque.retval(Deque()) \rightsquigarrow Exception$   
 $size(Deque()) = 0; \quad size(push(x, e)) = size(x) + 1$

**Fig. 2.** An algebraic specification of the Deque data abstraction

According to the classification scheme of [1], a method can be a *constructor*, an *observer* or a *modifier*. A constructor is a method that produces a new instance of the class. An observer is a method that returns a values expressing some information about the internal state of the object (e.g., the size of a container), while a modifier is a method that changes the internal state of the object. In practice, a method can play both roles of observer and modifier. It is therefore useful to distinguish between *impure* and *pure* observers; that is, observers that modify the internal state or not, respectively. In the case of the *Deque* data abstraction on Fig. 1, the method *Deque()* is a constructor; method *size()* is a pure observer, method *push(String)* is a modifier and methods *pop()* and *deq()* are both observers and modifiers.

Section 2.1 briefly introduces algebraic specifications, while Section 2.2 illustrates intensional behavior models. Thus, we proceed to introduce how those models can be compared.

## 2.1 Algebraic Specifications

Algebraic specifications (ASs), initially investigated in [5], are nowadays supported by a variety of languages, such as [12]. ASs model a component's hidden state implicitly by specifying axioms on sequences of operations. An algebraic specification  $\Sigma = (\Pi, E)$  is composed of two parts: the *signature*  $\Pi$  and the *set of axioms*  $E$ . Formally, a signature  $\Pi = (\alpha, \Xi, F)$  is a tuple where  $\alpha$  is the sort to be defined by the specification,  $\Xi$  is the set of the external sorts, and  $F$  is a set of functional symbols  $f_i$ , describing the signatures of operations. Each functional symbol has a *type*, that is, a tuple  $t \in (\{\alpha\} \cup \Xi)^+$ . The length  $n_t$  of each  $t$  specifies the arity of the functional symbol; the first  $n_t - 1$  elements specify the domain of the functional symbol while the last element denotes its range; we denote each functional symbol as  $f_i : \xi_1, \dots, \xi_{n_t-1} \rightarrow \xi_{n_t}$  (where  $\xi_i \in (\{\alpha\} \cup \Xi)$ )

to clearly distinguish domain and range. Each axiom is a universally quantified formula expressing an equality among terms in the algebra. Fig. 2 shows the AS for our illustrating example, the *Deque* described in Fig. 1. The notation used in this specification explicitly manages the case of impure observers by using two different implicitly defined operations, one for the returned value (e.g., the *pop.retval* operation) and one for the sort to be defined (e.g., the *pop.state* operation). Moreover, we model exceptions as particular values of the codomain of observers. In the case of impure observer, we specify the exception as a particular returned value and we state that its occurrence does not modify the internal state. For example, see the definition of the impure observer *pop* in Figure 2.

In this paper, we consider a particular class of ASs, called *linear specifications*. An algebraic specification is linear when its signature is linear. A signature defining a sort  $\alpha$  is *linear* if the following conditions hold: (i) there is exactly one constant  $f : \rightarrow \alpha$ ; (ii) Every non-constant function is in the form  $f : \alpha, \xi_2, \dots, \xi_{n_t} \rightarrow \xi_{n_t}$ , with  $\xi_{i < n_t} \in \Xi$  and  $\xi_{n_t} \in (\{\alpha\} \cup \Xi)$ . For simplicity, we also require axioms to not include hidden functions and conditional axioms. The set of axioms defines the properties that the specified data abstraction should exhibit. Formally, the concept of *algebra* is used to assign semantics to signatures and specifications. An algebra is composed of a set, the *carrier set* of the algebra, and a family of functions on that set. An algebra  $A$  is a  $\Pi$ -algebra, that is, it satisfies the signature  $\Pi$ , if it gives an interpretation of the sorts and the functional symbols in the signature. Moreover, an algebra  $A$  is also a  $\Sigma$ -Algebra, that is, it satisfies the whole specification  $\Sigma$ , if  $A$  gives an interpretation of the signature  $\Pi$  which also satisfies the set of axioms. The actual semantics prescribed by the set of axioms depends on the semantics given by the equality relation of them. Given a possible implementation of the data abstraction adhering to a specification  $\Sigma$ , which is by definition a  $\Sigma$ -algebra, the equality relation expressed with the set of axioms can be interpreted as a specification of which sets of instances are in the same *abstract state* [1]. Different definitions of this concept exist in the literature. The most commonly used is based on the concept of *behavioral equivalence* [13]. Given two objects  $o_1$  and  $o_2$  instances of a class  $C$ ,  $o_1$  and  $o_2$  are *behaviorally equivalent* if for any sequence of operations  $t$  of  $C$  ending with an observer, the objects  $o_1.t$  and  $o_2.t$  obtained by invoking  $t$  are themselves behaviorally equivalent. For observers returning primitive types, they are behaviorally equivalent if their values are the same.

HEUREKA [8] is a tool for recovering ASs for JAVA classes. HEUREKA leverages on the concept of behavioral equivalence to infer which sequence of method invocations produce instances that are likely to be behaviorally equivalent. Thus, the equations produced by this step are generalized into likely algebraic axioms.

## 2.2 Intensional Behavior Models

Another possible way to specify the behavior of stateful components is by using *behavior models*. Essentially, a behavior model is a finite state automaton where each state is labeled with observer return values and each transition represents a modifier invocation. *Behavioral equivalence models* (BEM) [7] are particular

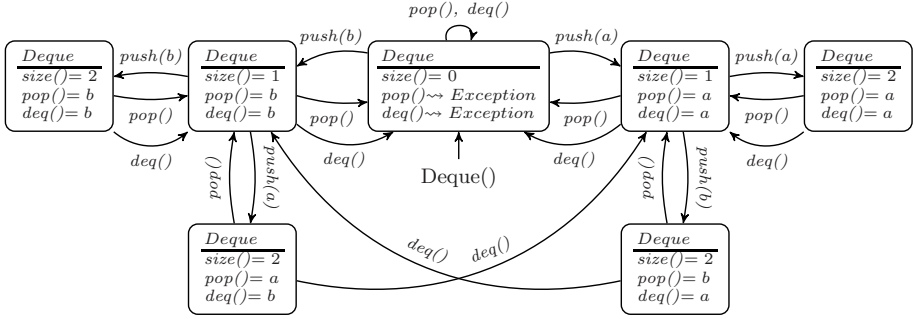


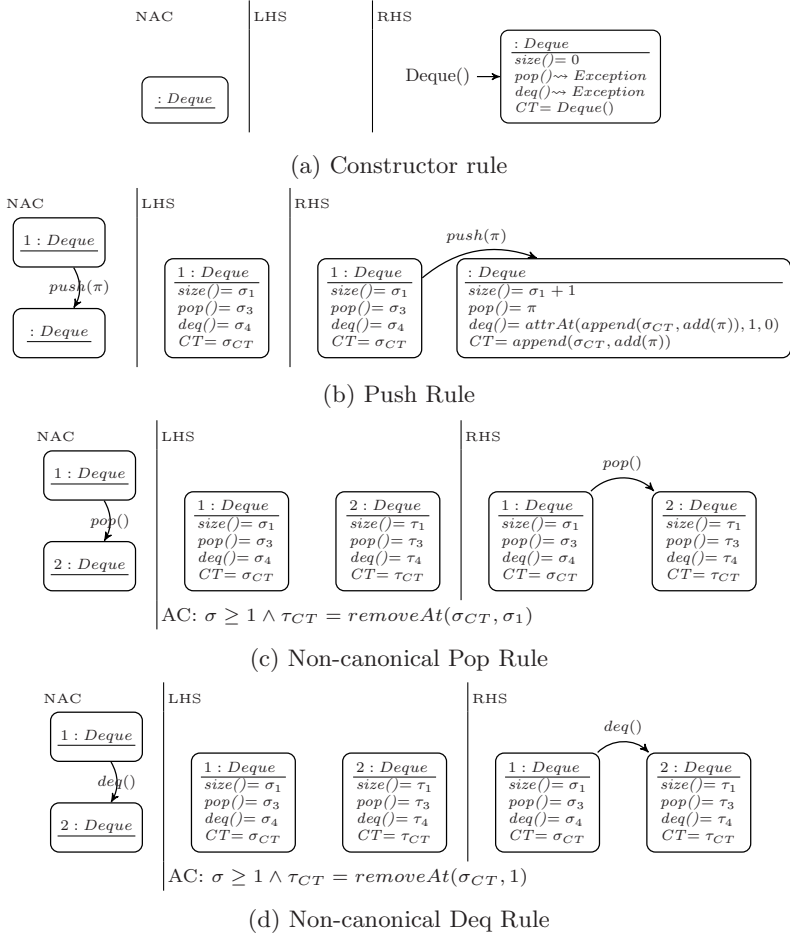
Fig. 3. A BEM of the Deque container

kind of behavior models, where each state represents a set of behaviorally equivalent instances of a data abstraction. A BEM is defined by choosing a finite set of actual parameters for each method. Figure 3 shows a possible BEM for the *ArrayDeque* data abstraction when *a* and *b* are used as actual parameters for the *push* method. Thus, each transition modeling the behavior of the *push* method is labeled with either *push(a)* or *push(b)*. Each state is labeled with observer return values. Obviously, a finite state machine cannot describe every possible behavior of the *Deque* data abstraction, even if we limit the inserted elements to two possible strings. For example, the BEM of Figure 3 models the behavior of the data abstraction only up to size 2.

To overcome this limitation, we proposed *intensional behavior models* [9], and a corresponding recovery technique, called SPY [7]. The key idea is to intensionally describe every possible BEM of the data abstraction. Since BEMs are finite-state automata, they can be viewed as graphs, with nodes labeled with observer return values. *Attributed graph transformation systems (GTS)* [14] can be used to intensionally describe the generation of a set of attributed graphs. In this way, we can specify how to generate all possible BEMs corresponding to all possible instances of the container class of interest.

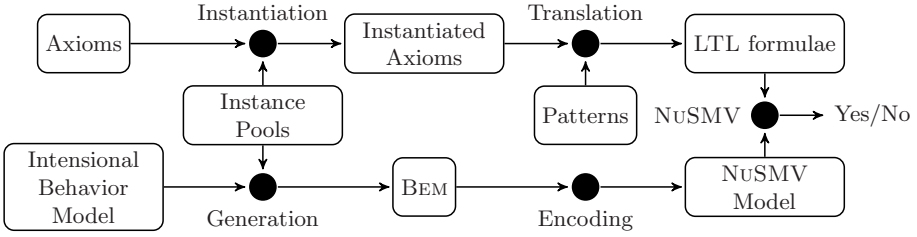
A GTS is composed of a set of rules, as in a classic Chomsky grammar. In a GTS, rules describe how a graph is modified by their application. Each rule is described by three graphs, the *negative application condition (NAC)*, the *left hand side graph (LHS)*, and the *right hand side graph (RHS)*, and a set of attribute conditions *AC*. Figure 4 describes the intensional behavior model of the *Deque* data abstraction. A rule can be applied when the following conditions hold for a source graph. The *LHS* describes which topological conditions must be matched by a subgraph of the source graph to make the rule applicable. The application of the rule replaces such subgraph with the subgraph described by the *RHS*. *NACs* express conditions that must not be matched for the rule to be applied. Both *LHS* and *NAC* nodes and arcs are labeled with variables on the domain of attributes. The *AC* set is composed of binary predicates on variables defined on the *LHS* attribute variables.

For example, Figure 4(a) describes the rule for the constructor of the *Deque* data abstraction. Consider an initial empty graph. The *LHS* of the constructor



**Fig. 4.** Deque Intensional Behavior Model

rule is trivially matched because the graph is empty and the *LHS* is empty, and the *NAC* is not matched since the empty graph does not contain any *Deque* node. Thus, the constructor rule is applicable. The *RHS* graph describes how the matching subgraph must be modified if the rule is applied; in the case of the constructor rule, it introduces an initial *Deque* node representing the empty deque. As for the *push* rule of Figure 4(b), note that integer numbers are used to establish a correspondence between nodes of *LHS* and *RHS*. If the applicability conditions are verified, the rule transforms the source graph into a new graph. The resulting graph is built by replacing the *LHS* subgraph with *RHS*. Numbered nodes in the *LHS* are replaced by identically numbered nodes in *RHS*. Attributes are modified according to functions labeling nodes in the *RHS*. Referring to the *push* rule in Figure 4(b), the application of *RHS* adds a new state representing the state obtained after a *push* application, and a



**Fig. 5.** Outline of the Validation Approach

transition labeled with the modifier. The newly introduced state represents a new set of behaviorally equivalent instances built through sequences of applications of operations. The *NAC* prevents further applications of the *push* rule with the same *LHS* and the same string as a parameter. If we have two String objects on the graph representing the instance pool to generate a BEM, the rule can be applied two times, generating two states representing a stack with a single element. It is then possible to apply the rule in Figure 4(d), corresponding to a *pop* method, by matching states 1 and 3 of the rule with the empty stack and the stack containing a single element.

The SPY [7] tool implements intensional behavior model recovery from dynamic analysis. It exploits the same notion of behavioral equivalence as HEUREKA. It first starts by recovering a BEM for the class to be analyzed, and then tries to generalize its transitions by recovering intensional behavior model rules. We do not include details on the recovery approach in this paper; the reader who is interested in more details can refer to [7].

### 2.3 Outline of the Validation Approach

We now provide an intuitive description of the foundations of the proposed validation approach. The aim of the proposed method is to validate an AS against an intensional behavior model, both modeling the behavior of a container. Figure 5 describes a workflow of the proposed approach through its constituent steps. In an ideal world, given a specification  $\Sigma$  and an intensional behavior model  $I$ , it would be desirable to check that the (possibly *infinite*-state) BEM generated by  $I$  satisfies the specification  $\Sigma$ , that is, the set of states, together with the transition function and the state labelling are a  $\Sigma$ -algebra. However, we already emphasized that containers have a state space that may be infinite, and in general not even denumerable, as in the case of the *Deque* data abstraction over the set of all possible strings on a finite alphabet.

We therefore limit the analysis as follows. First, we require the analyzer to provide interpretations of the external sorts in the ASs; we require that those sorts, together with the operations among them, have a finite carrier set. Those external sorts define the so-called *instance pools*, that is, the set of actual parameters for methods, that are used to generate a BEM from the intensional behavior



model. As already stated, limiting the instance pools to be finite does not imply that the container has a finite number of states; for this reason, we also limit the rule application to generate a finite-state BEM. In this case, verifying the consistency of the specification  $\Sigma$  to such a finite-state model, boils down to verifying that the algebra determined by those limitations is a  $\Sigma$ -algebra. Since we want to verify that the algebra determined by the BEM is a  $\Sigma$ -algebra, we must precisely interpret the symbols of the specification with the mathematical definition of the BEM. Let us consider a linear specification  $\Sigma = (\Pi, E)$ . A BEM over the same signature  $\Pi$  is a tuple  $\mathcal{B}_\Pi = \langle Q, I, \delta, q_0, \Psi \rangle$ , composed of a set of states  $Q$ , an initial state  $q_0$ , an input set  $I$ , a transition function  $\delta$ , and a set  $\Psi$  of state labelling functions representing observer return values. The input set of the BEM is the set of instantiated modifiers  $I = \bar{\mathcal{M}}_\Pi$ . Moreover, the set  $\Psi$  is composed as follows: for every observer functional symbol  $f_o : \alpha, \xi_1, \dots, \xi_{n_t-1} \rightarrow \xi_{n_t} \in F$ , there is a  $\Psi_{f_o} : Q \times \bar{\mathcal{O}}_\Pi \rightarrow \xi_{n_t}$ , which is a state labelling function representing return values for the set of instantiated observers  $\bar{\mathcal{O}}_\Pi$ .

The sets of instantiated modifiers  $\bar{\mathcal{M}}_\Pi$  and observers  $\bar{\mathcal{O}}_\Pi$  are defined as follows. Let us consider the provided instance pools,  $IP(\xi_i)$ , for each external sort  $\xi_i \in \Xi$ . Thus, the sets of instantiated modifiers and observer return values are defined as tuples whose first element is the functional symbol and the other elements are elements from the instance pools. For a given modifier  $f_m : \alpha, \xi_1, \dots, \xi_{n_t-1} \rightarrow \alpha \in F$ , the possible invocations of the modifier with the specified instance pools are:  $\bar{\mathcal{M}}_{f_m} = \{\langle f_m, e_1, \dots, e_{n_t} \rangle \mid e_i \in IP(\xi_i)\}$ . As of the *push* method, the instantiated modifiers with  $IP(String) = \{a, b\}$ , are  $\langle push, a \rangle$  and  $\langle push, b \rangle$ . The whole set of possible modifier invocations, that is, the input set of the BEM, is the union of the instantiated modifiers for each modifier:  $\bar{\mathcal{M}}_\Pi = \bigcup_{f_m} \bar{\mathcal{M}}_{f_m}$ . Similarly, we define instantiated observer set  $\bar{\mathcal{O}}_{f_o}$  for a given observer  $f_o$  and the whole set of possible observer invocations  $\bar{\mathcal{O}}_\Pi$ . We are now ready to define  $A(\mathcal{B}_\Pi)$ , that is, the  $\Pi$ -algebra over the BEM:

- The carrier sets for each external sort  $\xi_i \in \Xi$  are the instance pools  $IP(\xi_i)$ ;
- The carrier set for the defined sort  $\alpha$  is the set of states  $Q$ ;
- For each functional symbol  $f \in F$  of the signature  $\Pi$ , we defined the interpretation  $f^{A(\mathcal{B}_\Pi)}$  as follows:
  - Since  $\Pi$  is linear, there is only one constructor  $f_c$ , for which  $f_c^{A(\mathcal{B}_\Pi)} = q_0$ ;
  - For every modifier  $f_m : \alpha, \xi_1, \dots, \xi_{n_t-1} \rightarrow \alpha \in F$ ,  $f_m^{A(\mathcal{B}_\Pi)} = \delta|_{I \in \bar{\mathcal{M}}_{f_m}}$ ;
  - For every observer  $f_o : \alpha, \xi_1, \dots, \xi_{n_t-1} \rightarrow \xi_{n_t} \in F$ ,  $f_o^{A(\mathcal{B}_\Pi)}(q, e_1, \dots, e_{n_t-1}) = \Psi_{f_o}(q, \langle e_1, \dots, e_{n_t-1} \rangle)$ .

At this point, the core of our approach relies on validating the axioms of the specification  $\Sigma$  over the BEM  $\Pi$ -algebra  $A(\mathcal{B}_\Pi)$ . Given the interpretations provided by the BEM  $\Pi$ -algebra, axioms can be rewritten accordingly, and they become simple properties of the transition and labeling function of the BEM. Let us consider the simple BEM on Figure 3, and consider the following axiom:  $\forall s \in Deque, e \in String : pop(push(s, e)).state = s$ . With the chosen instance pools and the given interpretation, the axiom becomes:  $\forall s \in Q, e \in IP(String) : \delta(\delta(s, \langle push, e \rangle), \langle pop \rangle) = s$ . Since  $IP(String)$  is finite, the axiom

can be instantiated for every external sort:  $\forall s \in Q, \delta(\delta(s, \langle push, a \rangle), \langle pop \rangle) = s \wedge \delta(\delta(s, \langle push, b \rangle), \langle pop \rangle) = s$ .

The last step is the precise definition of the validity of axioms in our model. The only quantified values in this case can be elements of the specified sort, that is, states of the BEM. Theoretically, it would be possible to verify directly those axioms by proving that for every possible valuation of variables in the model. However, since the BEM is finite, the interpreted functions might be partial, and thus most of the axioms could be not verified just because the model is finite. Instead, we would like to verify that in all the cases on which the interpreted functions are defined on the BEM, the axioms hold. For example, the axiom above cannot hold in any finite BEM of the *Deque*, since there does not exist a finite BEM where the *push* operation is defined in every state. Thus, our problem reduces to verifying the axioms in all the valuations of the variables for which the transition function  $\delta$  is defined, and we consider the axiom holding precisely in these cases. Fortunately, an explicit management of this problem can be avoided by a proper encoding of the BEM, which will be clear in the following section.

### 3 Validating Axioms through Model Checking

In the previous section, consistency of an algebraic specification with an intensional behavior model has been reduced to determining if the axioms of an algebraic specification  $\Sigma$  are verified in the BEM  $\Pi$ -algebra  $A(\mathcal{B}_\Pi)$ . Axioms can be interpreted as properties of the transition relation  $\delta$  and the observer labeling functions  $\Psi_{f_o}$ . In this section, we will show how this problem can be reduced to checking temporal logic formulae derived from the algebraic axioms, as they are interpreted in the BEM  $\Pi$ -algebra  $A(\mathcal{B}_\Pi)$ , against a Kripke structure derived by the BEM. To prove this, we encode the BEM as a Kripke structure and the property over the infinite traces as an LTL formula. The approach is realized in two steps, which correspond to the structure of this section:

1. *Formal BEM encoding*: the BEM is encoded into a Kripke structure;
2. *Axiom rewriting*: algebraic axioms are translated into temporal formulae expressed in LTL, by following certain translation patterns.

**Formal BEM Encoding.** A BEM is encoded into a Kripke structure which can be directly used to generate an equivalent model in the input language of the NuSMV model checker. A Kripke structure is similar to a nondeterministic finite state automaton, where each state is labeled with a set of atomic propositional formulae  $\Phi$ . Formally, the Kripke structure is composed of a *Frame*  $\mathcal{F}$  and an evaluation function  $V$ . The frame is a tuple  $\mathcal{F} = \langle \mathcal{S}, \mathcal{S}_0, \mathcal{R} \rangle$ , where  $\mathcal{S}$  is a set of states,  $\mathcal{S}_0 \subseteq \mathcal{S}$  are the initial states of the frame, and  $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S}$  is the reachability relation between states. The evaluation function  $V : \Phi \rightarrow \wp(\mathcal{S})$  essentially defines which atomic formulae are true in which states. In fact, for each formula  $\phi \in \Phi$ ,  $V(\phi)$  is the set of states where  $\phi$  is true. Our encoding of a BEM as a Kripke structure prescribes that each state of the frame corresponds to a state of the BEM together with an operation, that is, to a pair  $\langle q, i \rangle$  with

$q \in Q \wedge i \in I$ . Formally, the set  $\mathcal{S}$  of states of the frame is defined as  $\mathcal{S} = \{\langle q, i \rangle : q \in Q \wedge i \in I \wedge \exists q' \in Q : \delta(q, i) = q'\}$ ; the set of initial states is defined as  $S_0 = \{\langle q, i \rangle : \langle q, i \rangle \in \mathcal{S} \wedge q = q_0\}$ . In other words, each state on the frame models a state of the BEM where an existing transition, corresponding to a given modifier, is enabled. Thus, we encode the reachability relation as follows:  $\langle q, i \rangle \mathcal{R} \langle q', i' \rangle \Leftrightarrow \delta(q, i) = q'$ . This encoding is a classic way to translate a deterministic transition-labeled automaton into a Kripke structure where non-deterministic models the possible operation choice. Practically, from each state of the frame, the next reachable states always represent the state reached by applying a given transition on the BEM, but they differ with respect to the next possibly enabled operation.

In almost any existing model checker, the frame is defined by means of some temporal logic axioms or some operational constructs, whose semantics implicitly defines the structure of the frame, that is, its states and its reachability relation. Thus, the encoding is based on the direct use of the set of atomic formulae  $\Phi$  and axioms based on them. We can split the set of atomic formulae  $\Phi$  in three different sets to encode the Kripke structure defined above:

- $\Phi_S$ , the set of atomic formulae representing states of the BEM;
- $\Phi_I$ , the set of atomic formulae representing enabled transitions;
- $\Phi_O$ , the set of atomic formulae representing observer return values.

$\Phi_S$  has the same cardinality as the set of states  $Q$  of the BEM. It contains a set of mutually exclusive propositions, each being true iff the current state of the frame models a given state of the BEM. If we define a representation bijection  $\mu_S : \Phi_S \rightarrow Q$ , then  $\forall \phi_S \in \Phi_S : V(\phi_S) = \{\langle \mu_S(\phi_S), i \rangle \in \mathcal{S}\}$ . The same encoding is applied to transitions (i.e., instantiated modifiers). That is, if we define a representation bijection  $\mu_I : \Phi_I \rightarrow I$ , then the evaluation function  $V$  is defined as follows:  $\forall \phi_I \in \Phi_I : V(\phi_I) = \{\langle q, \mu_I(\phi_I) \rangle \in \mathcal{S}\}$ . Finally, we must encode observer return values with ad-hoc propositional formulae. Consider any observed pair of instantiated observer and return value present in the BEM:  $\langle \bar{o}, e_{n_i} \rangle$ , such that  $\bar{o} \in \bar{O}_{f_o} \wedge \exists q \in Q : \Psi_{f_o}(q, \bar{o}) = e_{n_i}$ . For any of these pairs, we define a specific propositional formula, defined by a representation bijection  $\mu_O$ . Then,  $\forall \phi_O \in \Phi_O : \mu_O(\phi_O) = \langle \bar{o}, e_{n_i} \rangle \Rightarrow V(\phi_O) = \{\langle q, i \rangle \in \mathcal{S} : \Psi_{f_o}(q, \bar{o}) = e_{n_i}\}$ . In practice, we encode each observer - return value pair in the BEM as an atomic formula in the Kripke structure. At this point, we have defined every possible atomic formula used in the encoding of the BEM. As stated above, we need to encode the frame structure described previously as a set of axioms. For space reasons, we omit the actual encoding; the reader can find them in [15].

**Rewriting Axioms as LTL Formulae.** To explain the rationale behind the translation of algebraic axioms into LTL formulae, consider the axiom  $\forall x \in Deque, e \in String : pop.state(push(x, e)) = x$  and its equivalent property on the  $\delta$  function derived by interpreting the BEM:  $\forall s \in Q, e \in IP(String) : \delta(\delta(s, \langle push, e \rangle), \langle pop \rangle) = s$ . Consider all the possible infinite traces of the BEM, that is, all the  $\omega$ -words defined in the alphabet  $\bar{\mathcal{M}}_{\Pi}$ , and the generalized transition function  $\delta^* : Q \times \bar{\mathcal{M}}_{\Pi}^* \rightarrow Q$ . Suppose that the axiom above holds in the

model. Given any infinite trace  $x \in \bar{\mathcal{M}}_I^\omega$ , if the axiom holds, then for every finite prefix  $\bar{x}$  of  $x$  such that  $\bar{x} = \bar{x}_0 \langle push, a \rangle \langle pop \rangle$  or  $\bar{x} = \bar{x}_0 \langle push, b \rangle \langle pop \rangle$ , then the state reached by the sequence of operations  $\bar{x}_0$  is the same as the one reached by  $\bar{x}$ , that is,  $\delta^*(q_0, \bar{x}_0) = \delta^*(q_0, \bar{x})$ . In practice, the traces we are interested in all the ones generated by the transitions where the  $\delta$  function of the BEM has been defined. The encoding we defined above guarantees that the traces generated by the Kripke structure are exactly those. Each state of the frame encodes both the current state of the BEM, that is, a formula in  $\Phi_S$ , and the current operation applied, that is, a formula in  $\Phi_I$ , together with formulae encoding the current return values of observers. Thus, a property about the traces of the BEM model like the one we defined for the axiom above, can be written as an LTL formula. In the case above, the corresponding LTL formula is:  $\forall \phi_S \in \Phi_S : G(\phi_S \wedge \mu_I(\langle push, a \rangle) \wedge X(\mu_I(\langle pop \rangle))) \Rightarrow X^2 \phi_S \wedge G(\phi_S \wedge \mu_I(\langle push, b \rangle) \wedge X(\mu_I(\langle pop \rangle))) \Rightarrow X^2 \phi_S$ . A pattern for axioms in this form is the following:

**Pattern 1.** *Any axiom in the following form:  $\forall x \in \alpha : m_j(\dots m_1(m_0(x)) \dots) = x$  where  $m_i, n_i \in \bar{\mathcal{M}}_I$ , is translated to the following LTL formula:  $\forall \phi_s \in \Phi_S : G(\phi_s \wedge \mu_I^{-1}(m_0) \wedge X(\mu_I^{-1}(m_1)) \wedge \dots \wedge X^j(\mu_I^{-1}(m_j))) \Rightarrow X^{j+1} \phi_s$*

Please note that a formal proof of the correspondence expressed by this pattern cannot be included for space limitations. However, the reader can find proofs in [15]. We identified several patterns for translating algebraic axioms to LTL formulae, based on the approach described above, but for space reasons we are not able to show all the patterns.

## 4 Evaluation

Both the NUSMV encoding of the BEM and the algebraic axiom translation have been implemented as a software tool. We now proceed to empirically evaluate the performance of the encoding and model checking as prescribed by the proposed approach. In Section 1, we illustrated one of the possible applications of our validation approach in the context of specification recovery. In this paper, we proposed a solution to an instance of the general problem of automatic comparison of recovered formal specifications of containers, that is, the problem of checking algebraic specifications against intensional behavior models. Moreover, the tools and the extracted specifications are particularly suitable for classes implementing containers. Thus, an empirical assessment may compare algebraic specifications and intensional behavior models as recovered from the respective extraction tools, for relevant containers such as the ones implemented in the JAVA library. We selected a set of container classes implemented in the *java.util* package of the JAVA library, and extracted both algebraic specifications with HEUREKA and the intensional behavior models with SPY. We report here two different evaluation experiments. For both the experiments, we generated a random set of instance pools to instantiate the algebraic axioms and generate a BEM of the class to be analyzed. In the first experiment, we used the same set

**Table 1.** Empirical Results

Class	Experiment I				Experiment II				
	Axioms		Performance		Axioms		Performance		
	Ver.	Not Ver.	Time (mm:ss)	Mem. (MB)	Ver.	Not Ver.	Time (mm:ss)	Mem. (MB)	
<b>ArrayDeque</b> 32 states	Exh.	20	0	00:04.30	63.68	10	3	00:04.30	52.4
	BMC	1	0	00:11.4	74.40	0	1	00:01.12	44.40
<b>PriorityQueue</b> 37 states	Exh.	13	0	00:03.50	30.01	7	4	00:02.10	23.01
	BMC	1	0	00:14.40	57.92	0	0	—	—
<b>Stack</b> 157 states	Exh.	11	0	00:07.90	68.68	4	0	00:02.90	40.58
	BMC	1	0	00:09.40	60.92	1	0	00:09.40	60.92
<b>TreeMap</b> 64 states	Exh.	21	0	07:06.00	274.87	12	6	04:06.00	204.87
	BMC	1	0	01:00.20	269.26	0	0	—	—
<b>TreeSet</b> 33 states	Exh.	23	0	03:05.20	69.64	13	7	01:35.20	49.64
	BMC	3	0	41:42.50	511.12	1	0	12:21.53	317.14

of test cases as inference basis for both the extraction methods. The inference basis was manually checked to be relevant in the sense that it included all the interesting behaviors of the component; the testing approach was similar to the simpler one used to assess the SPY method [7]. The rationale behind this choice is that we expect the two specifications to be coherent. Instead, the second experiment uses on purpose two different inference bases. We choose to use the same inference basis of the first experiment for the intensional model, and instead use a smaller inference basis for HEUREKA. The smaller inference basis has been chosen to not include some behaviors of the component. We expect that some of the recovered algebraic axioms could be wrong; thus, some of the properties expressed by these axioms should not be verified by the model checker. The reason for this choice is simply to verify that our approach is able to detect inconsistency behind recovered specifications. Table 1 shows empirical results of the validation approach for both the experiments. The first column include the name of the checked class and immediately below the number of states of the generated BEM. The second column contains the number of axioms that have been checked, showing verified and not verified axioms. The last two columns include the total time and memory needed for the verification. The experiments have been performed in a Intel® Core Duo™ machine at 2.16 Ghz with 2 Gb of RAM. In the general case, we tried to verify each axiom with the exhaustive search, based on BDDs [10]. For each class, the first row of the table illustrates the empirical results for axioms for which the exhaustive verification was possible under reasonable amounts of time (i.e., within an hour of execution time). In some cases, especially with patterns involving two sequences of operations, exhaustive search could be too expensive in terms of execution time and memory consumption, up to unfeasibility with the used resources. For such axioms, we

used the *Bounded Model Checking (BMC)* [16] feature of NUSMV, based on SAT solving techniques. Essentially, BMC limits the search up to a given depth. Results of the first experiment on Table 1 show that every axiom has been verified in the intensional behavior model, that is, that the specification recovered by HEUREKA does not contradict the model inferred by SPY. Instead, the results of the second experiment show that some of the axioms were not verified, and thus our approach is able to detect inconsistencies between algebraic specifications and intensional behavior models.

## 5 Related Work

This paper proposed a methodology to cross-validate intensional behavior models and algebraic specifications by using model checking. The use of model checking is justified because it is inherently a methodology for cross-validation of specifications. In fact, model checking consists in general in the problem of checking if an operational description satisfies a set of properties expressed in temporal logic. Both the operational description and the temporal logic properties can be considered as specifications, and the process of model checking can be seen as a method to validate their consistency. In particular, a recent advance [17] introduces *multi-valued model checking*, which explicitly manages situations like uncertainty and inconsistency. Other related works come from the algebraic specification community; for example, HETCASL [18] is a framework for the formal analysis of heterogeneous algebraic specifications by means of theorem proving. Several related state-of-the-art techniques come from requirements engineering community, that is, from methodologies involving discovery and management of inconsistent requirements within the context of multiple viewpoints or requirement-related artifacts. A recent advance [19] proposed goal model checking over operational descriptions derived from scenarios. Finally, some relevant related works include techniques for model comparison to support software evolution analysis, such as [20]. However, those techniques are used to compare the same software artifacts during their evolution, and not to compare different software artifacts.

## 6 Conclusions

We illustrated a method for comparing specifications of classes implementing containers by using model checking. In particular, we proposed a model-checking based technique to check the consistency of intensional behavior models against algebraic specifications. In fact, the former can be used to generate a particular finite-state model, the behavioral equivalence model, while the latter plays the role of a set of properties to be verified. To perform model checking, we provided a formal encoding of the BEM as a Kripke structure and a practical encoding in the source code of the NUSMV model checker. Moreover, we identified a comprehensive set of patterns to translate algebraic specifications to LTL formulae. We showed that it is possible to check algebraic specifications against intensional behavior models in reasonable amounts of time and occupied memory.

## References

1. Guttag, J.V., Liskov, B.: Program Development in Java: Abstraction, Specification and Object-Oriented Design. Addison-Wesley, Reading (2001)
2. Ghezzi, C., Jazayeri, M., Mandrioli, D.: Fundamentals of Software Engineering. Prentice Hall PTR, Upper Saddle River (2002)
3. Harel, D., Gery, E.: Executable object modeling with statecharts. In: ICSE 1996: 18th International Conference on Software Engineering (1996)
4. Leavens, G.T., Baker, A.L., Ruby, C.: JML: A notation for detailed design. In: Behavioral Specifications of Businesses and Systems, pp. 175–188. Kluwer, Dordrecht (1999)
5. Guttag, J.V., Horning, J.J.: The algebraic specification of abstract data types. *Acta Informatica* 10(1) (1978)
6. Ernst, M.D.: Dynamically Discovering Likely Program Invariants. Ph.D. thesis, University of Washington, Seattle, Washington (August 2000)
7. Ghezzi, C., Mocci, A., Monga, M.: Synthesizing intensional behavior models by graph transformation. In: ICSE 2009: Proc. of 31st Int. Conf. on Soft. Eng. (2009)
8. Henkel, J., Reichenbach, C., Diwan, A.: Discovering documentation for Java container classes. *IEEE Trans. Software Eng.* 33(8), 526–543 (2007)
9. Baresi, L., Ghezzi, C., Mocci, A., Monga, M.: Using graph transformation systems to specify and verify data abstractions. In: Proc. of GT-VMT 2008 (2008)
10. Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.J.: Symbolic model checking:  $10^{20}$  states and beyond. *Inf. Comput.* 98(2)
11. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. MIT Press, Cambridge (2006)
12. Mosses, P.D. (ed.): CASL Reference Manual. LNCS (IFIP Series), vol. 2960. Springer, Heidelberg (2004)
13. Doong, R., Frankl, P.G.: The ASTOOT approach to testing object-oriented programs. *ACM Trans. on Soft. Eng. and Meth.* 3(2) (1994)
14. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. EATCS Monographs in TCS. Springer, Heidelberg (2005)
15. Spy Checker Website (2009), <http://home.dei.polimi.it/mocci/spy/check/>
16. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic model checking without BDDs. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, pp. 193–207. Springer, Heidelberg (1999)
17. Chechik, M., Devereux, B., Easterbrook, S., Gurfinkel, A.: Multi-valued symbolic model-checking. *ACM Tr. Softw. Eng. Methodol.* 12(4) (2003)
18. Mossakowski, T., Maeder, C., Lüttich, K.: The heterogeneous tool set, hets. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 519–522. Springer, Heidelberg (2007)
19. Uchitel, S., Chatley, R., Kramer, J., Magee, J.: Goal and scenario validation: a fluent combination. *Requir. Eng.* 11(2), 123–137 (2006)
20. Ivkovic, I., Kontogiannis, K.: Tracing evolution changes of software artifacts through model synchronization. In: ICSM 2004. IEEE Computer Society, Los Alamitos (2004)