Electronic Communications of the EASST
Volume X (2008)

Proceedings of the
Seventh International Workshop on
Graph Transformation and Visual Modeling Techniques
(GT-VMT 2008)

Using Graph Transformation Systems to Specify and Verify Data Abstractions

Luciano Baresi, Carlo Ghezzi, Andrea Mocci and Mattia Monga

14 pages

Guest Editors: Claudia Ermel, Reiko Heckel, Juan de Lara
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Using Graph Transformation Systems to Specify and Verify Data Abstractions

Luciano Baresi[1], Carlo Ghezzi[2], Andrea Mocci[3] and Mattia Monga[4]

{ baresi[1], ghezzi[2], mocci[3] } @elet.polimi.it
DeepSE Group
Dipartimento di Elettronica e Informazione - Politecnico di Milano
Piazza L. Da Vinci, 32 20133 Milano, Italy

[4] mattia.monga@unimi.it,
DICo - Università degli studi di Milano
Via Comelico, 39 20135 Milano, Italy

Abstract: This paper proposes an approach for the specification of the behavior of software components that implement data abstractions. By generalizing the approach of behavior models using graph transformation, we provide a concise specification for data abstractions that describes the relationship between the internal state, represented in a canonical form, and the observers of the component. Graph transformation also supports the generation of behavior models that are amenable to verification. To this end, we provide a translation approach into an LTL model on which we can express useful properties that can be model-checked with a SAT solver.

Keywords: Graph Transformation Systems, Specifications, Data Abstractions, Model Checking, SAT Solving

1 Introduction

Abstraction by specification [LG00] is the fundamental approach to abstract from implementation details by providing a high-level description of the behavior of a software component. Such an abstract description of the behavior is the *specification* of the software component.

In this paper, we focus on the specification of *data abstractions* [LG00], which can be viewed as a particular class of stateful components. In a data abstraction, the internal state is hidden; clients can interact with the component by invoking the operations exposed by its *interface*. Common examples of data abstractions are stacks, queues, and sets, which are usually part of the libraries of modern object oriented languages, such as JAVA.

Several formalisms have been proposed in the past for the specification of data abstractions. Among these, we mention the pioneering work on algebraic specifications [GH78, GTW78]. Recently, in the area of recovering specifications and program analysis, behavior models (see for example [DLWZ06, XMY06]) have been proposed as a simple formalism to relate the observable part of a data structure with the methods used to modify its internal state. In this paper, we propose a generative approach for the construction of behavior models based on graph transformation systems. Moreover, we found that the generative capabilities of graph transformation

tools are suitable for the generation of models that can be easily translated into logic models for verification. For this purpose, we propose a translation of graph-transformation generated models into a linear temporal logic, on which verification is possible via bounded model checking.

This paper is organized as follows. Section 2 recalls some background concepts on the specification of data abstractions. Section 3 presents the case of a traversable stack, a container whose specification has been critically analyzed in the past because of its subtle intricacies. Section 4 describes our approach to the specification of stateful components implementing data abstractions. Section 5 contains our bounded model checking approach to the verification. Finally, Section 6 concludes the paper.

2 Formalisms for Data Abstractions

Data abstractions hide their internal state and implementation and export a set of operations (methods), to allow clients to access their instances. Methods can be classified as:

- *constructors*, which produce a new instance of a class;
- *observers*, which return some view of the internal state;
- *modifiers*, which change the internal state.

A method might both modify the internal state and return some information about it. In this case, such an observer is called *impure*; otherwise, it is *pure*, that is, it has no side-effects and does not alter the internal state of the object.

Stateful components implementing data abstractions may be specified by providing pre- and post-conditions [Hoa69] for the methods exposed by the component's interface. Because a method may also change the internal state, this approach requires the introduction of a logical abstraction of the internal state, which complicates the specification of the component. For example, the JML language [LBR99] uses *models* for this purpose.

Algebraic specifications do not generally require an explicit abstraction of the hidden state, since it is implicitly taken into account through the use of axioms on sequences of operations. An algebraic specification is composed of two parts: the *signature* and the *set of axioms*. The algebraic signature defines the types (*sorts*) used in the set of axioms, and the signatures of the operations. The set of axioms is usually composed of a sequence of universally quantified formulae expressing equalities among terms in the algebra. For example, an algebraic specification for a class implementing a stack of strings may be characterized by the following axiom:

$$\forall s \in Stack, \forall e \in String : pop(push(s,e)) = s$$

which states that for every possible stack, the object obtained by the application of a push followed by a pop is equivalent to the original object. Algebraic specifications are supported nowadays by various languages and tools [GH93, Com04].

2.1 Behavior Models

A behavior model is a finite-state automaton that captures the relationship among the modifiers and return values of the observers. In a behavior model, each state is an abstraction of a set of

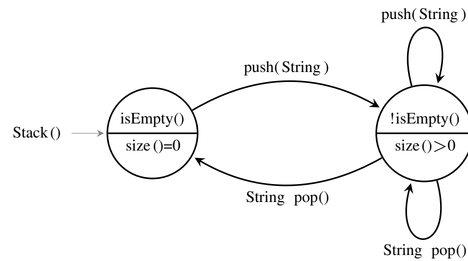


Figure 1: A simple abstracted behavior model of a Stack.

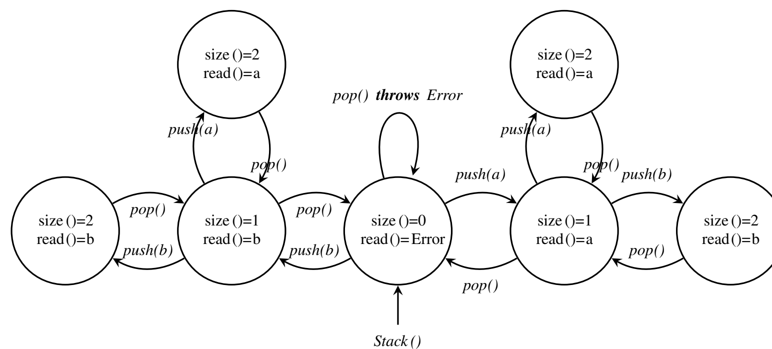


Figure 2: A deterministic behavior model for a Stack

different internal states of the actual component, identified by a simple predicate on the return values of a set of observers. Several different approaches to recover behavior models have been implemented for various purposes. For example, ADABU [DLWZ06] behavior models are characterized by states labeled with pure observers, while transitions are labeled with method names (without actual parameters). Other approaches, such as ABSTRA [XMY06], produce more precise finite state machine abstractions by using different techniques to represent the component's state. Figure 1 shows the simple behavior model of a stack produced by ADABU; in this model, the behaviors of all the stacks with size greater than zero are abstracted in a unique state.

Since infinitely many states are summarized in a finite-state machine, the resulting automaton is necessarily non-deterministic. However, a deterministic model is often more useful to reason about the properties of the component or to generate test cases; thus, we aimed at producing a deterministic specification, that is, one in which the application of each operation brings the automaton to a unique state, which represents a unique state of the object.

Figure 2 shows a partial deterministic model of the behavior of a stack. Deterministic models grow bigger and, for components with infinite possible internal states, they can only partially represent the behavior of the component. Deterministic models, such as the one of Figure 2, rely on the presence of a set of *instance pools* for the actual parameters and represent the behavior of the component for such parameters. In the example, the *push* method is tested with an instance pool for the input parameter composed of two different strings, "a" and "b". In this paper, we present

```
public class MTStack {
  public MTStack() { .. }
  public void push(String element) { .. }
  public void pop() throws Error { .. }
  public void down() { .. }
  public void reset() { .. }
  public String read() throws Error { .. }
  public boolean isEmpty() { .. }
  public int size() { .. }
}
```

Figure 3: The public interface of Majster's Traversable Stack

an approach to the representation of the complete, unabstracted and thus deterministic behavior of stateful components defining data abstractions by using graph transformation systems.

3 Traversable Stack

In this section we introduce an example of data abstraction that is used as a running example to explain our approach. The example is inspired by a case study that generated a lively debate in the late 1970s in the community working on algebraic specifications. We will refer to the example as Majster's Traversable Stack (MTS) [Maj77], by the name of the author who first addressed the problem. The public interface of MTS is shown in Figure 3. For simplicity, we assume the contained object to be of type `String`. MTS defines the usual operations of a stack, such as *push* and *pop*, and allows for traversal by using a hidden pointer and by exposing the following operations:

- *down*, which traverses the stack circularly by moving the pointer stepwise towards the bottom of the stack;
- *read*, which yields the element returned by the pointer;
- *reset*, which moves the pointer on the top of the stack.

For example, let us consider a stack of three elements, obtained by applying the constructor and three *push* operations of three different elements, the strings "a", "b" and "c". In this case, the *read* observer returns "c". If a *down* operation is applied, the hidden pointer moves towards the bottom of the stack; thus, the *read* observer returns "b". If the hidden pointer reaches the bottom of the stack, a further application of the *down* operation brings the hidden pointer to the top of the stack.

MTS was introduced because the author argued that no finite set of axioms could specify the data abstraction in an algebraic way without using auxiliary functions, that is, purely in terms of the externally visible operations. For example, the specification of the *down* operation would require axioms like the following:

$$down^n\left(down^{n-1}\left(\ldots\left(down^1\left(push^n\left(\ldots push^2\left(push^1(MTStack(),o_1),o_2\right)\ldots\right),o_n\right)\right)\ldots\right)\right) = push^n\left(push^{n-1}\left(\ldots push^2\left(push^1(MTStack(),o_1),o_2\right)\ldots,o_{n-1}\right),o_n\right)$$

Since n [1] is generic in the axiom formula, the specification would require an axiom for each $n > 0$.

In the sequel, we will show how MTS can be rigorously specified in our approach, based purely on the operations exported by the data abstraction's interface.

4 A Graph Transformation approach

In this section we illustrate our approach to the specification of data abstractions based on graph transformations. Let us consider the behavior model for a stack depicted in Figure 2: it describes the behavior of all the stacks up to size 2. The specification of an abstract data type is given as a graph transformation system whose rules can be applied not only to generate such a partial (deterministic) model but also any other model that, for example, describes the behavior of stacks up to a generic size n.

In this paper we use the approach to graph transformation systems implemented in AGG [Tae04], which supports a rich set of features, such as negative application conditions, attributes defined as JAVA objects, and conditions on attributes. In the following, we define the *type graph* for MTS, which defines the kinds of graph nodes and edges that are needed in the specification, and then we define the graph transformation rules.

4.1 Type graph

The type graph for MTS is shown in Figure 4. A state of the behavior model is either a *null state* or an *object state* (a stack of strings). Conventionally, a null state represents the state of an instance object of the data abstraction before any application of a constructor. The *null state* is unique and represents the initial state of the behavior model. The type node representing an object state is labeled with a set of attributes, which represent the return values of the observers invoked when the object is in the corresponding state. Edges represent constructors and modifiers. Constructors link the null state node with a node representing the object state after applying a constructor, while modifiers are loops on the state node since they change the state of the instance. Each edge is labeled with attributes corresponding to the parameters, the return value, and the exception the method might rise when applied (exceptions are modeled as Boolean values).

As already said, behavior models rely on instance pools of parameters to be used to generate actual invocations of modifiers. For example, we can generate a behavior model for MTS that uses two strings, "a" and "b" as the possible contained objects. The type graph has a node for each type needed, labeled in the same way as the data abstraction node. For primitive types and

[1] In this and in the following formulae, the superscript j above each operation indicates the j-th subsequent operation of that kind.

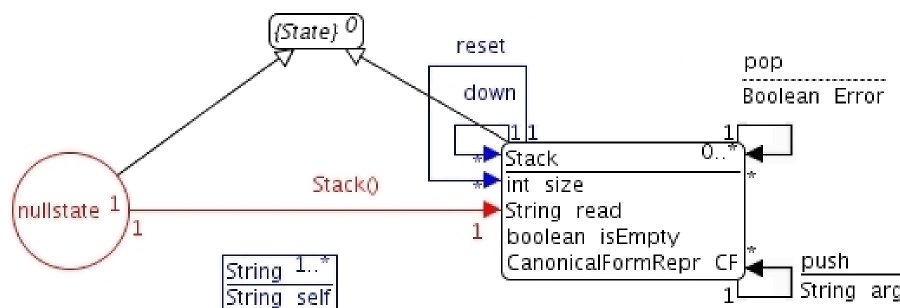


Figure 4: MTS Type Graph

Strings, we can *box* the value in a node that contains just an attribute representing the primitive type.

4.2 Rules and Canonical Form

Since data abstractions hide their internal state, the information gathered by just invoking observers might be insufficient to uniquely identify the object state. For example, let us consider two MTS of size 2, containing the same string "a" on the top and two different strings on the bottom, "a" for the first stack and "b" for the second. In both cases, let us consider the hidden pointer to be on the top of the stack. The invocation of the three observers (*read*, *size*, *isEmpty*) of the MTS is insufficient to reveal the different internal state, since they would return the same values (i.e., "a" for *read*, 2 for *size* and **false** for *isEmpty*).

Inspired by Veloso's algebraic specification of MTS [Vel79], we use a *canonical form* for the data abstraction to identify each different object state. We define the canonical form as a language composed by method applications as tokens. The canonical form language must satisfy the following properties:

- each string of the language is composed of an initial constructor and a —possibly empty— sequence of modifiers;
- for each possible internal state, there is one and only one string of the canonical form language to represent it;
- for each string of the language, there is an internal state that is labeled by that string, such that the invocation of the corresponding sequence of methods produces an object of that state.

As a convenience, we identify a language of operations, which satisfies these properties, as a canonical form of the data abstraction. This explains why the object state node of the type graph on Figure 4 is enriched with an attribute (CF) describing its canonical form. According to this approach, any possible MTS instance can be represented with a string of the following canonical form language $\mathscr{L}_{MTS}$:

- ε, which conventionally labels the null state;

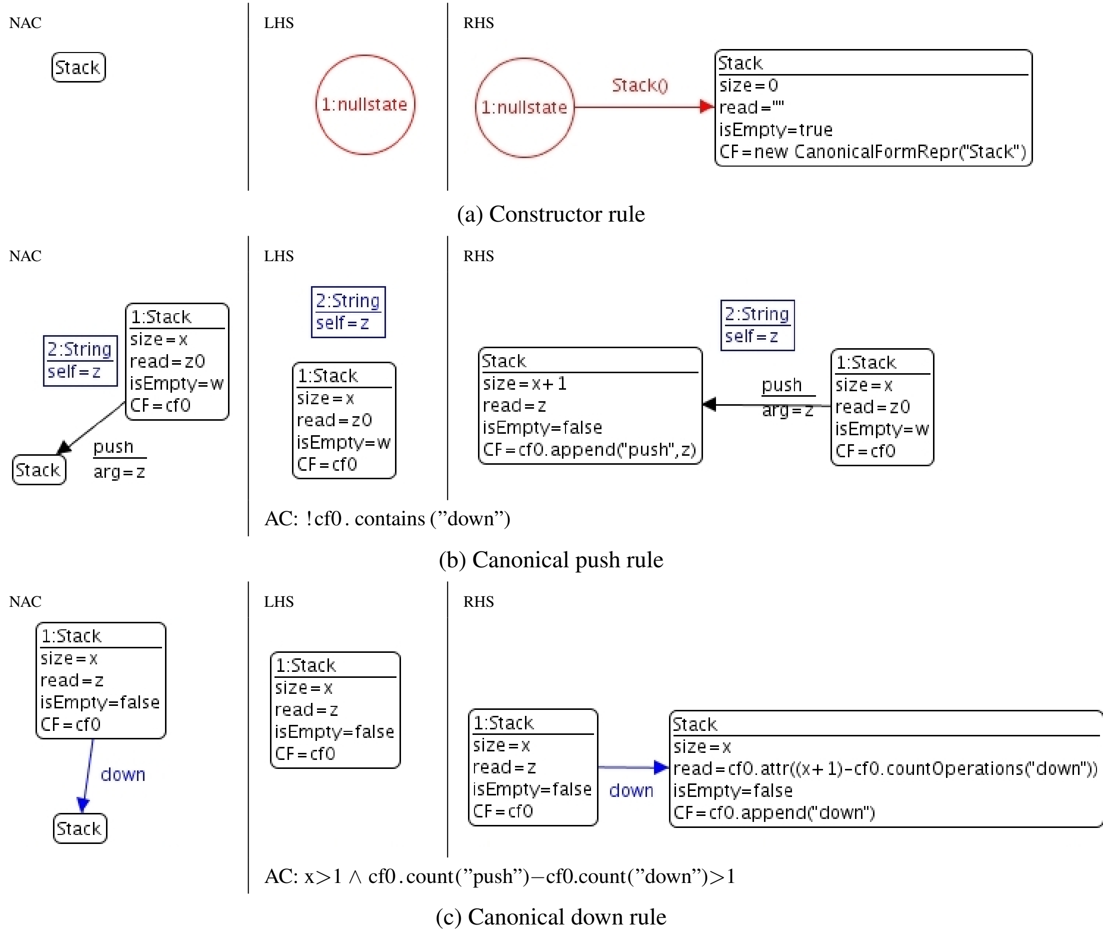


Figure 5: Partial GT specification of MTS (part I)

- $MTStack()$, that is, the empty stack after calling the empty constructor;
- $push^n\left(push^{n-1}\left(\ldots push^1\left(MTStack(), o_1\right)\ldots, o_{n-1}\right), o_n\right)$, that is, a stack after any non-empty sequence of *push*es;
- $down^m\left(down^{m-1}\left(\ldots\left(push^n\left(\ldots push^1\left(MTStack(), o_1\right)\ldots, o_n\right)\right)\ldots\right)\right), 1 \leq m < n$, that is, any non-empty and non-singleton stack with at least one down invocation but strictly less than the stack size.

For each internal state of the instances of the data abstraction, and thus for each string of the canonical form language, we can define which operations on that state are *canonical*, and those that are not. Given a state X labeled with a canonical form $\mathscr{X} \in \mathscr{L}_{MTS}$, an operation m is canonical in X if $\mathscr{X} \cdot m \in \mathscr{L}_{MTS}$.

For example, in the case of the canonical form of MTS described above, the language defines when a *push* operation has to be considered as canonical, that is, whenever no (canonical) *down* operations have been applied to the instance. Every other operation is *non-canonical*: such

operations bring the data abstraction in a state for which a corresponding different canonical string in the canonical language already exists. For example, the chosen canonical form language $\mathscr{L}_{MTS}$ implies that every *pop* operation is non-canonical.

Once the language has been identified, we can define GT rules dealing with canonical operations (explained in Section 4.3) and other rules dealing with non-canonical operations (explained in Section 4.4. For example, the language of canonical method applications defined in Section 3 for MTS can be used as a canonical form language for the specification of the data abstraction.

Since for each state of the instances there exists a corresponding string of the canonical form language, we can use it to label each state of the graph. We implemented the canonical form attribute type with an ad-hoc JAVA class that stores a list of strings, each representing an application of a canonical operation. Each string is stored together with all the actual parameters of the corresponding operation application, which can be accessed by invoking the observers on the instance of the canonical form.

4.3 Canonical Form Rules

A canonical form rule defines a state generation, i.e., it specifies how a new state in a canonical form can be generated from an existing one by applying a canonical operation. Canonical form rules share the following common template:

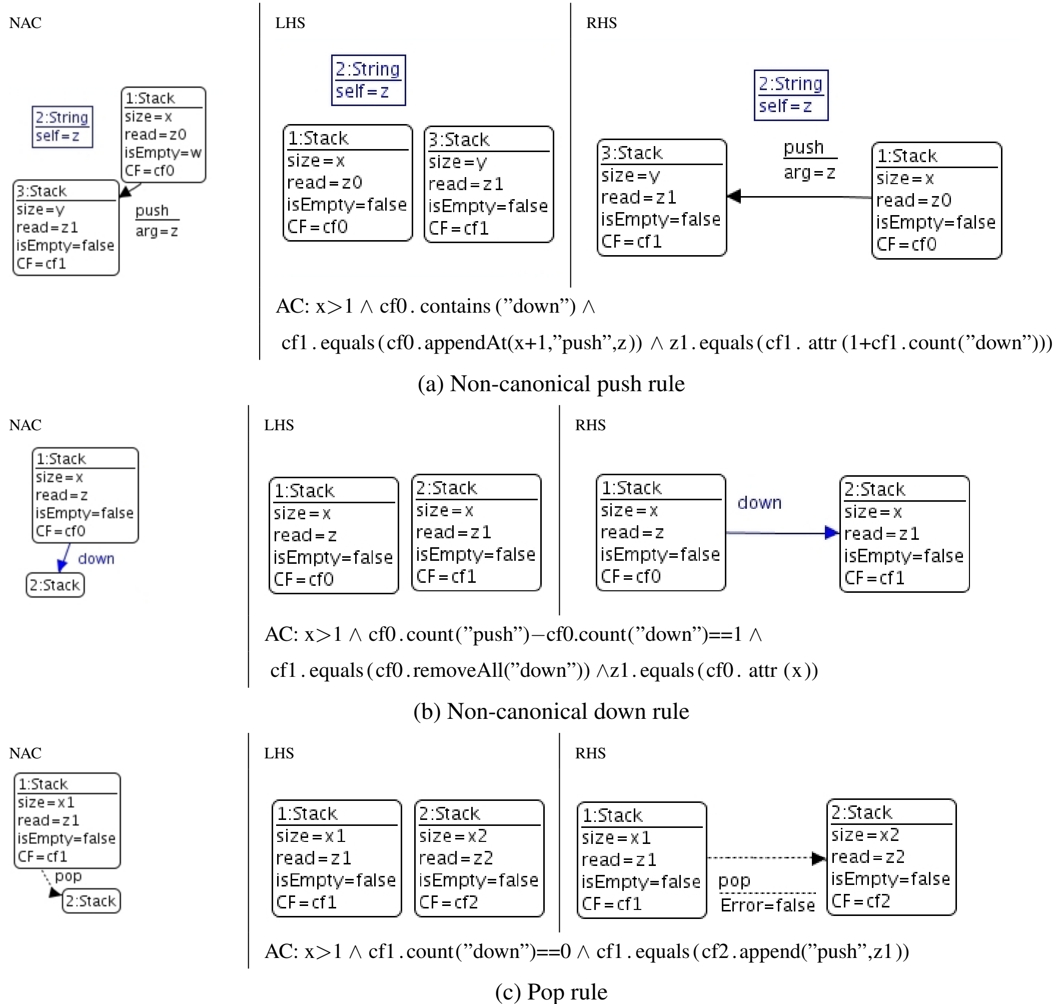
- The Left-Hand Side (LHS) of the rule is always composed of a single node for the data abstraction state, and a set of nodes representing the parameter's values needed for applying the canonical operation;
- The Right-Hand Side (RHS) preserves the nodes contained in the LHS and, furthermore, it adds the new canonical state and a new edge between the two data abstraction states, to represent the application of the canonical operation.

For example, the MTS requires a canonical rule for the default constructor, and other rules for the canonical applications of methods *push* and *down*.

Canonical constructor rules generate new transitions from the null state to new nodes representing the object state after the invocation of a constructor. The LHS of constructor rules is composed of the null state and a set of nodes representing the parameter values for the invocation of the constructor. The RHS adds the generated state and initializes the observer attributes. The canonical form representation of the generated node is initialized with a new value representing the application of the constructor. For example, MTS exposes only one constructor, which can be chosen as part of the canonical form. The canonical constructor rule is shown in Figure 5a.

Modifier rules differ from constructor rules just because the node in the LHS is a node representing an object state: it cannot be the null state. The node must be identified by a set of attribute conditions, which uniquely identify the correct state on which the rule must be applied.

Figure 5b shows the rule for operation *push*. Attribute conditions state that the canonical form cf0 must not contain any *down* operation. The rule introduces a new state, with *size* increased by one, the *read* observer returning the argument of the last *push* operation, and the canonical form modified by appending the *push* operation. Figure 5c shows the canonical *down* rule, which can be applied whenever the difference between the number of *push* operations and that of *down*



(a) Non-canonical push rule

(b) Non-canonical down rule

(c) Pop rule

Figure 6: Partial GT specification of MTS (part II)

operations in the canonical form is greater than one. The introduced state has the same *size* as the previous and the *read* method returns the first attribute of the jth element of the normal form, where j = (x + 1) − cf0. countOperations ("down"). It is not hard to prove that with the defined canonical form language that element in the canonical form is always a *push* and that after each *down* operation the *read* method returns the corresponding element obtained by moving the hidden pointer towards the bottom of the stack. Negative Application Conditions (NACs) preserve the determinism of the generated behavior model.

4.4 Non-Canonical Method Rules

The generative approach of canonical constructors and modifiers rules defines the state space of the data abstraction. Thus, we represent non-canonical method applications as rules that add edges between existing nodes of the graph. With this approach, non-canonical method rules can

be applied iff the related states have been already generated by canonical form rules. The LHS and RHS of a non-canonical method rule must contain the two nodes, and the RHS of the rule creates the edge between them that represents the application of the method.

In the MTS case, method *pop* is not part of the canonical form. Thus, every transition corresponding to the application of a pop operation is added by non-canonical modifier rules (see Figure 6c). A more complex case regards operations *down* and *push*. In fact, these operations are not always part of a canonical form, depending on the context of the previous canonical operations applied to the object. For example, operation *down* is canonical only when applied after a sequence of $n > 1$ *push*es, starting with a constructor, and for a maximum of $n-1$ times. Otherwise, it is not canonical. Thus, a non-canonical rule (see Figure 6b) for operation *down* must be added to handle the non-canonical case. The non-canonical rule has a different context of application, represented by a different condition on attributes. In this case, the non-canonical *down* moves the hidden pointer to the top of the stack. In a similar way, the *push* operation is not canonical when applied after a *down* operation. In Veloso's specification, the operation is valid, and we can specify it with the rule of Figure 6a.

The MTS specification does not contain any example of non-canonical constructor rule. Such a rule would be similar to the case of non-canonical modifiers.

4.5 Canonical Form and Topology

In principle, attribute conditions on the canonical form could be expressed by topological patterns. For example, given the canonical push rule of Figure 5b, one can notice that the attribute condition that states that the canonical form does not contain any *down* operation could be expressed by the following informal topological constraint: there exists a path from the null state to the node on the LHS of the rule, starting with a constructor edge and composed only of push edges. Such topological conditions cannot be expressed with AGG, thus we used equivalent attribute conditions.

5 Verification

Our specification can be used to verify interesting properties of data abstraction specifications. Our verification approach has two steps: (1) AGG is used to generate a behavior model of the data abstraction; (2) the resulting model is translated into a Linear Temporal Logic (LTL) model, on which we apply SAT-solving [DP60] to model-check properties.

5.1 Model Generation

In general, model checking a specification expressed with graph transformations is not easy; several solutions have been proposed for the general case [BS06, RSV04]. However, the behavior of our rules, as expressed in the previous section, is limited: they can only add new nodes or edges to the graph. At each transformation step, the graph on which the rule is applied is left unmodified by the rule itself. For this reason, the model to be checked is an instance graph (i.e., a behavior model of the data abstraction) after a limited number of rule applications, and it can be generated by simply using AGG.

Graph transformation rules can generate infinitely many behavior models. For example, for any data abstraction with potentially infinite states, our specification approach can generate an unbounded number of graphs representing the behavior of the data abstraction. To model check the specification against certain properties we need to bound generated models by adding some attribute conditions on the canonical form rules. Non-canonical form rules are implicitly limited by the fact that they operate by only adding edges between existing nodes.

For example, suppose that we want to verify a given property on the MTS previously defined; we might choose to limit the scope of the verification to all the possible states of a MTS with $size < 4$. We can limit the application of canonical rules, and thus the size of the model, by adding this constraint as a constraint to the attributes of canonical form rules. In the case of MTS, the canonical rule to be constrained is the *push* canonical rule. If the size attribute of the node on the LHS is greater than or equal to 4, the *push* canonical rule is not applied, thus stopping the generation of new nodes.

5.2 LTL Translation

The model obtained by applying the rules of the graph grammar is translated into an LTL description which can be fed to ZOT [PMS07], together with the set of properties that should be satisfied by all the possible states of the data abstraction. ZOT produces a corresponding set of propositional formulae which can be automatically used as input for a SAT-solver.

The axioms for the logical model are built in the following way:

- We define a variable with finite domain for the state of the behavior model; the domain is a set of items defining the state in which an object of the data abstraction can be.
- Similarly, we define a set of variables for each observer attribute on each node representing a state of the object; for example, we might define a variable named *size* for the corresponding observer that can assume a set of different integer values, such as $\{0,1,2,3\}$ if we correspondingly limit the model.
- We define a variable called *do* that represents the name of the operation enabled in the transition; it can assume a set of values composed of the names of the constructors and modifiers, together with a string representation of their actual parameters.

Axioms A set of axioms is needed to characterize each state with the corresponding observer variables. Thus, the translation contains a set of implications of the following kind:

$$state = s_{1a} \Rightarrow (size = 1) \wedge (isEmpty = false) \wedge (read = a)$$

Enabled transitions can be expressed by defining axioms that denote which operations can be applied in a state. For example, the following axiom:

$$state = s_0 \quad \Rightarrow \quad (do = push_a \vee do = push_b \vee do = pop_E)$$

states that the only possible operations that can be done on the empty stack are *push* and *pop* operation, which is exceptional.

Finally, we need axioms to define the postcondition of the application of methods. For example, the axiom:

$$do = stack \Rightarrow X(state = s_0)$$

states that the next state after the invocation of the constructor is the state corresponding to the empty stack. Another example is given by the following implications:

$$(state = s_0 \wedge do = push_a \Rightarrow X(state = s_{1a})) \wedge (state = s_0 \wedge do = push_b \Rightarrow X(state = s_{1b}))$$

These implications define the different behavior obtained by pushing different elements on the empty stack. With all these axioms defined, we have an LTL specification on which each transition represents the application of an operation on instances of the data abstraction. For this reason, we can use the LTL operator X to predicate on which states are reached after the application of a sequence of methods.

5.3 Example Properties

Let us consider the following simple property, which states that the application of a *pop* after a *push* in any state brings the object to the same initial state:

$$\forall x \in state_vals \quad ((state = x) \wedge (do = push_a \vee do = push_b) \wedge X(do = pop) \\ \Rightarrow X^2(state = x))$$

Precisely, the property states that if we are in state x and we *push* a or b, and then we do a *pop*, the final state is again x. Another, more complex property, for the MTS is that for any state x where the size of the stack is greater than or equal to 2, the application of $n = size()$ *down* operations brings the object to state x:

$$\forall x \in state_vals, y \in \{2,3\} \quad ((state = x) \wedge (size = y) \wedge \big(\forall k (0 \leq k < y \Rightarrow X^k(do = down))\big) \\ \Rightarrow X^y(state = x))$$

We fed ZOT with the LTL translation of a generated behavior model with states with $size < 4$, and checked it against these two properties. The LTL model, together with the negated properties, was found unsatisfiable in a few seconds; thus, the two properties are valid for the chosen bound.

6 Conclusions

¡¡¡¡¡¡¡ .mine This paper presented an approach to the specification and verification of data abstractions by using graph transformations. The generative nature of graph transformations provided a means to describe intensionally the (potentially infinite) behavior models of abstract data types, thus overcoming the limitations of plain finite-state automata. Graph transformations can be used to generate behavior models of the specified data abstraction which can be

easily translated to a finite set of axioms suitable for automatic verification, for example by using a SAT-solver. By using a generative approach, the complexity of the generated model can be tailored to the verification needs. ======= ¿¿¿¿¿¿¿ .r799

¡¡¡¡¡¡¡ .mine

We provided the specification of traversable stack, and showed how some interesting properties can be verified with our approach. Our ongoing work is now trying to infer behavior models defined through graph transformations by observing execution traces of existing module libraries.

======= This paper presents an approach to the specification and verification of data abstractions by using graph transformations. The generative nature of graph transformation provided a means to describe intensionally the (potentially infinite) behavior models of abstract data types, thus overcoming the limitations of plain finite-state automata. Graph transformations can be used to generate behavior models of the specified data abstraction which can be easily translated into a finite set of axioms suitable for automatic verification, for example by using a SAT-solver. By using a generative approach, the complexity of the generated model can be tailored to the verification needs.

Our ongoing work is now on trying to infer behavior models defined through graph transformation systems by observing execution traces of existing module libraries.

¿¿¿¿¿¿¿ .r799

Bibliography

[BS06] L. Baresi, P. Spoletini. On the Use of Alloy to Analyze Graph Transformation Systems. In Corradini et al. (eds.), *ICGT 2006: Proceedings of 3rd International Conference on Graph Transformation*. Lecture Notes in Computer Science 4178, pp. 306–320. Springer, 2006.

[Com04] Common Framework Initiative. CASL *Ref. Manual*. LNCS 2960. Springer, 2004.

[DLWZ06] V. Dallmeier, C. Lindig, A. Wasylkowski, A. Zeller. Mining Object Behavior with ADABU. In *WODA 2006: Proceedings of 4th International Workshop on Dynamic Analysis*. May 2006.

[DP60] M. Davis, H. Putnam. A Computing Procedure for Quantification Theory. *J. ACM* 7(3):201–215, 1960.

[GH78] J. V. Guttag, J. J. Horning. The Algebraic Specification of Abstract Data Types. *Acta Informatica* 10(1), 1978.

[GH93] J. V. Guttag, J. J. Horning. *Larch: languages and tools for formal specification*. Springer-Verlag New York, Inc., New York, NY, USA, 1993.

[GTW78] J. A. Goguen, J. W. Thatcher, E. W. Wagner. An Initial Algebra Approach to the Specification, Correctness and Implementation of Abstract Data Types. In *Current Trends in Programming Methodology*. Volume 4, pp. 80–149. Prentice Hall, 1978.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *CACM* 12(10):576–580, 1969.

[LBR99] G. T. Leavens, A. L. Baker, C. Ruby. JML: A Notation for Detailed Design. In Kilov et al. (eds.), *Behavioral Specifications of Businesses and Systems*. Pp. 175–188. Kluwer Academic Publishers, 1999.

[LG00] B. Liskov, J. Guttag. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*. Addison-Wesley, Boston, MA, USA, 2000.

[Maj77] M. E. Majster. Limits of the “algebraic” specification of abstract data types. *ACM SIGPLAN Notices* 12(10):37–42, 1977.

[PMS07] M. Pradella, A. Morzenti, P. San Pietro. The symmetry of the past and of the future: bi-infinite time in the verification of temporal properties. In *ESEC-FSE ’07: Proceedings of the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. Pp. 312–320. ACM, New York, NY, USA, 2007.

[RSV04] A. Rensink, Á. Schmidt, D. Varró. Model Checking Graph Transformations: A Comparison of Two Approaches. In Ehrig et al. (eds.), *ICGT 2004: Proceedings of 2nd International Conference on Graph Transformation*. Lecture Notes in Computer Science 3256, pp. 226–241. Springer, 2004.

[Tae04] G. Taentzer. AGG: A Graph Transformation Environment for Modeling and Validation of Software. In Pfaltz et al. (eds.), *Application of Graph Transformations with Industrial Relevance*. LNCS 3062, pp. 446–456. Springer, 2004.

[Vel79] P. Veloso. Traversable stack with fewer errors. *ACM SIGPLAN Notices* 14(2):55–59, 1979.

[XMY06] T. Xie, E. Martin, H. Yuan. Automatic Extraction of Abstract-Object-State Machines from Unit-Test Executions. In *ICSE 2006: Proceedings of 28th International Conference on Software Engineering, Research Demonstrations*. Pp. 835–838. May 2006.