FAIL-STOP COMPONENTS BY PATTERN MATCHING

Tomasz Janowski and Wojciech I. Mostowski*
The United Nations University
International Institute for Software Technology
P.O. Box 3058, Macau, China
{tj,wim}@iist.unu.edu

Abstract We describe an approach to formally specify object-based software components, in order to be able to automatically check their behavior at run-time. The specification is a regular expression built from the propositions about the states (or pairs of states) of a component. Checking is done by a specification-generated wrapper, which produces a fail-stop component from a component which fails in an arbitrary way. The wrapper-generator is implemented for a subset of Java classes. We argue that specification-based error-detection is particularly suitable for the components of open, object-based distributed systems.

Keywords: Formal specifications, run-time checking, component wrapping, application generators, correctness by construction, fault-tolerance.

1. INTRODUCTION

Open object-based distributed systems challenge the traditional ways of applying formal methods via specification and proof. One of the problems is the large number of the components involved, which are partly decided at compile-time (static invocation) and partly at run-time (dynamic invocation). Another problem is having to rely on the vendor's claims about correctness of individual components, without being able (lacking the implementation details) to verify such claims ourselves. Yet another is expressing component specifications in an interface definition language, which describe how to communicate with a component (syntactic level), but not the expected results of such communication (semantics). Such problems make static verification difficult, at best.

*Institute of Mathematics, University of Gdańsk, Poland.

On the other hand, the structuring of the whole system in terms of the independent, distributed components, is particularly suitable for fault-tolerance [7]. The goal is to make sure that the failures of individual components (violation of their specifications) will not cause the whole system to fail (violation of the system's specification). The latter specifications can be used at design-time to prove if the system is indeed fault-tolerant. The former can be used at run-time to detect if a component has failed. This paper describes an approach to formally specify software components in order to make such error-detection possible.

Defining such specifications is not without problems. Specifications may contain infinite constructs like quantifiers (for all values of a type), liveness properties (for all states in a sequence) or modal properties of branching time (for all transitions from a state). Such constructs are generally non-executable – we cannot execute them directly on the machine, and non-checkable – we cannot check effectively at run-time that they indeed hold. On the other hand, specifications based on propositional logic are not enough expressive in practice. Finally, checking specifications which use equality of states is not possible when the state can be only accessed via defined operations; the best we can do is checking bisimilarity [9]. Such problems require a different kind of specification for effective run-time checking than those for static verification.

In this paper we propose a specification approach for software components which is suitable for their checking at run-time. The technical approach is as follows: (1) Specifications are formally-based. They are defined as logic-based regular expressions built from the propositions about the states (or pairs of states) of a component, via its observer operations. (2) Specifications are checkable at run-time, based on the recorded history of the component's execution, a sequence of observer values about individual states and the operations which caused state-changes. (3) Checking is carried out by a wrapper which is generated from the component and its specification. The wrapper takes over all communication between the component and its environment. It remains transparent to the clients except being able to detect (carrying out run-time checking after every state-change) and announce (via the additional boolean observer $error$) the occurrence of an error. (4) The required effect of run-time checking is described formally as the fail-stop property. The wrapper-generator is to transform a given component, which may fail in an arbitrary way, into a component which only fails by refusing to operate, also announces this fact. The formal model in RAISE [11] with justification of the correctness of the wrapper-generator (that every wrapped component is fail-stop), is the subject of a companion paper.

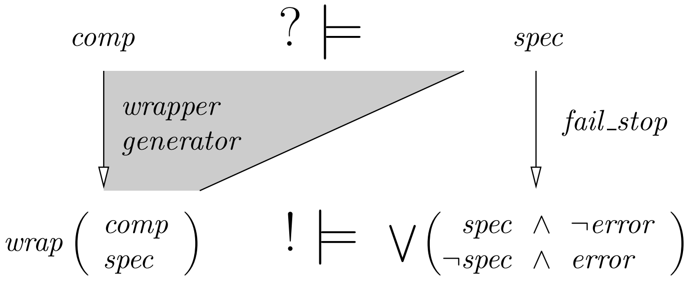

Figure 1 Fault-free versus fail-stop software components

Figure 1 provides an illustration. It describes the intended result of the wrapper generator for specifications given as state invariants. The component *comp* may or may not satisfy the invariant *spec* but its transformed version $wrap(comp, spec)$ satisfies the 'fail-stop' version of *spec*: *spec* holds if and only if *error* is false. Importantly, this property is not verified for each component and its specification but proved about the wrapper-generator, then applied to all components and specifications.

The rest of this paper is as follows. Section 2 explains and illustrates the concept of a 'component'. Section 3 discusses and compares 'fault-free' and 'fail-stop' behaviors of components. Section 4 shows how to ensure fail-stop behavior by patter-matching. Section 5 presents an example, a line editor. Section 6 describes how pattern-matching can become part of an automatically generated wrapper, also describes a prototype implementation for components written in Java. Section 6 provides some conclusions and directions for our future work.

2. COMPONENTS

We treat a component as a entity with its own state and the operations defined on it. The state is internal – the operations are the only means to give access to it, persistent – it maintains its value between operation invocations, and the operations are atomic – to execute them concurrently is the same as executing them in an arbitrary order.

We represent the state-space of a component as an abstract type *State* and its operations by the functions on this type. Depending on the type of access we divide the operations into 'readers' which return a value without modifying the state and 'writers' which change the current state without returning any value. Readers are further divided into 'constants' which return some value independent from the current state and 'observers' which return a value based on the current state. Writers are divided into 'generators' which return a new state, independent from the current one, and 'modifiers' which modify the current state. The class of an operation depends on the 'position' of *State* in its signature:

$$
\begin{array}{ll}
\mathit{constant}: & \mathit{Type} \to \mathit{Type} \\
\mathit{generator}: & \mathit{Type} \to \mathit{State} \\
\mathit{observer}: & \mathit{Type} \times \mathit{State} \to \mathit{Type} \\
\mathit{modifier}: & \mathit{Type} \times \mathit{State} \to \mathit{State}
\end{array}
$$

We assume that the generator *init* represents the initial state. In the simplest case the only types are *State* and *Bool*, with constants and observers as propositional variables. The first-order model adopts one more type, say *Nat*, and allows operations to take several arguments and return results of this type. In the higher-order model operations take several arguments and return results of different types. In this case *Type* represents *Bool*, *Nat*, Cartesian product of two types etc.

$$
\mathit{Type} \quad ::= \quad \mathit{Bool} \mid \mathit{Nat} \mid \mathit{Type} \times \mathit{Type} \mid \ \ldots
$$

As a simple illustration consider a component representing a stack. The component is represented by the class expression $Stack_1$ which contains definitions of types, signatures and axioms, as below.

$$
\begin{array}{l}
\mathit{Stack}_1 = \mathit{class} \\
\quad \mathit{type} \\
\quad\quad \mathit{State} = \mathit{Elem}^* \\
\quad \mathit{signatures} \\
\quad\quad\quad \begin{array}{ll}
\mathit{bnd}: & \mathit{Nat} \\
\mathit{init}: & \mathit{State} \\
\mathit{len}: & \mathit{State} \to \mathit{Nat} \\
\mathit{top}: & \mathit{State} \to \mathit{Elem} \\
\mathit{pop}: & \mathit{State} \to \mathit{State} \\
\mathit{push}: & \mathit{Elem} \times \mathit{State} \to \mathit{State}
\end{array} \\
\quad \mathit{axioms} \\
\quad\quad\quad \mathit{init} = \langle\rangle \\
\quad\quad\quad \mathit{len}(s) = \mathit{ln}(s) \\
\quad\quad\quad \mathit{len}(s) < \mathit{bnd} \Rightarrow \mathit{push}(e, s) = \mathit{con}(e, s) \\
\quad\quad\quad \mathit{len}(s) > 0 \Rightarrow \mathit{pop}(s) = \mathit{tl}(s) \wedge \mathit{top}(s) = \mathit{hd}(s) \\
\mathit{end}
\end{array}
$$

Elem is the type of elements we put on stack. The operations are: one constant *bnd* of type *Nat*, the maximum number of elements on stack; one generator *init*, the state of the empty stack; two modifiers *push* to push an element onto the stack and *pop* to pop the top element; and two observers *top* to return the top element and *len*, the number of elements on stack. The axioms define the actual behavior of the operations. The stack is implemented on the concrete state-space defined as a list of elements, with corresponding concrete definitions for operations *init*, *push*, *pop*, *top* and *len*. They apply the standard list functions *con* (construction), *hd* (head), *tl* (tail) and *ln* (length).

For the purpose of this paper we consider such components in isolation, as individual objects. On the other hand, to build an application we may need a richer component including a collection of objects [10]. This would be also the right level to introduce fault-tolerance, based on the error-detection on the lower level (individual objects).

3. FAULT-FREE VERSUS FAIL-STOP COMPONENTS

A generator is a typical algebraic operation to construct state values [13]. An observer is a co-algebraic operations for making the observations about the state [3]. A modifier is both an algebraic operation and co-algebraic. This casting of components into an algebra (a carrier set with functions into this set) or a co-algebra (a carrier set with functions from this set) provides the means to describe how the concrete execution of a component unfolds. Here we are interested in the opposite: how to abstract away from such concrete executions in order to describe how the component should behave in the first place.

For instance for the stack component we put forward five axioms: the length of *init* is zero, *pop* decrements and *push* increments the length, *top* returns the element which was recently pushed onto the stack and *pop* modifies the state back before the last push. All axioms except the last relate readers and writers, involving equality over Nat or $Elem$. The last relates two writers and involves equality over the type $State$.

$$
\begin{array}{l}
\mathit{Stack}_0 = \mathit{class} \\
\quad \mathit{type} \\
\qquad \mathit{State} \\
\quad \mathit{signatures} \\
\qquad \mathit{bnd}: \quad \mathit{Nat} \\
\qquad \ldots \\
\quad \mathit{axioms} \\
\qquad \mathit{len}(\mathit{init}) = 0 \\
\qquad \mathit{len}(s) > 0 \Rightarrow \mathit{len}(\mathit{pop}(s)) = \mathit{len}(s) - 1 \\
\qquad \mathit{len}(s) < \mathit{bnd} \Rightarrow \mathit{len}(\mathit{push}(e, s)) = \mathit{len}(s) + 1 \\
\qquad \mathit{len}(s) < \mathit{bnd} \Rightarrow \mathit{top}(\mathit{push}(e, s)) = e \wedge \mathit{pop}(\mathit{push}(e, s)) = s \\
\mathit{end}
\end{array}
$$

Once we described the component on the abstract ($Stack_0$) and concrete ($Stack_1$) levels, we should be able to verify that the concrete component is fault-free. The proof has to demonstrate that concrete definitions satisfy all axioms described by the specification, which boils down to proving first-order properties about concrete value domains. For instance, $len(s) < bnd \Rightarrow top(push(e, s)) = e$ has to first expand definitions of *push* and *top* and then apply a simple fact about lists.

$$
\begin{array}{l}
len(s) < bnd \Rightarrow top(push(e, s)) = e \\
len(s) < bnd \Rightarrow top(con(e, s)) = e \\
len(s) < bnd \Rightarrow hd(con(e, s)) = e \\
len(s) < bnd \Rightarrow e = e \\
len(s) < bnd \Rightarrow true \\
true
\end{array}
$$

However, this static verification is very often impossible, given the requirements for the availability of an implementation, its specifications and the feasibility to carry out the proof. The stack example was chosen to provide a demonstration with a minimum of the technical details, however such details would certainly complicate any more realistic proof.

Suppose instead of proving that the component is fault-free, we want to make sure that faults, after they occur, do not spread out uncontrollably. That we detect the errors as soon as they occur, and then make this fact known to the environment by a boolean observer *error*.

$$error : State \rightarrow Bool$$

We call such a component 'fail-stop'. A fail-stop component need not be fault-free but a fault-free component is certainly fail-stop, only it will never experience a failure and its *error* indicator is permanently set to $false$. We can verify that the component is fail-stop, with respect to a given specification, by proving that it is fault-free with respect to the weaker 'fail-stop specification'. We obtain such a specification by syntactic transformation from the original specification, adding *error* to its signatures and modifying the axioms to include the possibility of them being violated. We only require that the value of *error* is *true* in a given state iff at least one of the original axioms is violated in this state. When the axioms involve more than one state then we only require this property about the last state, provided none of the prior states have set *error* to *true*. $Stack_2$ below describes such a fail-stop version of $Stack_1$.

$$
\begin{array}{l}
\mathit{Stack}_2 = \mathit{class} \\
\quad \mathit{type} \\
\quad\quad \mathit{State} \\
\quad \mathit{signatures} \\
\quad\quad \begin{array}{ll} \mathit{bnd}: & \mathit{Nat} \\ \mathit{init}: & \mathit{State} \\ \mathit{error}: & \mathit{State} \rightarrow \mathit{Bool} \\ \ldots & \end{array} \\
\quad \mathit{axioms} \\
\quad\quad \mathit{true} \Rightarrow \\
\quad\quad\quad (\mathit{len}(\mathit{init}) = 0) \Leftrightarrow \neg \mathit{error}(\mathit{init}) \\
\quad\quad \neg \mathit{error}(s) \wedge \mathit{len}(s) < \mathit{bnd} \Rightarrow \\
\quad\quad\quad \left(\begin{array}{c} \mathit{len}(\mathit{push}(e,s)) = \mathit{len}(s) + 1 \\ \wedge \\ \mathit{top}(\mathit{push}(e,s)) = e \end{array} \right) \Leftrightarrow \neg \mathit{error}(\mathit{push}(e,s)) \\
\quad\quad \neg \mathit{error}(s) \wedge \mathit{len}(s) < \mathit{bnd} \wedge \neg \mathit{error}(\mathit{push}(e,s)) \Rightarrow \\
\quad\quad\quad \left(\begin{array}{c} \mathit{pop}(\mathit{push}(e,s)) = s \\ \wedge \\ \mathit{len}(\mathit{pop}(\ldots)) = \mathit{len}(\ldots) - 1 \end{array} \right) \Leftrightarrow \neg \mathit{error}(\mathit{pop}(\mathit{push}(e,s))) \\
\mathit{end}
\end{array}
$$

Although we could try proving directly that the component satisfies the fail-stop specification, this would not solve our problems, nor utilize the special form of such specifications. Instead, we would like to guarantee the satisfaction of the fail-stop specification by run-time checking. We now describes a specification method to make such checking possible.

4. FAIL-STOP COMPONENTS BY PATTERN-MATCHING

Run-time checking is only possible if the specification method allows it to be automated – checking automatically if a given behavior complies with the specification. However, specification methods are usually designed in order to be expressive, to enable abstraction and support effective reasoning, not to allow automatic checkability. In particular, we cannot check specifications which contain quantifiers (over infinite types) and we cannot check equality between states ($\mathit{pop}(\mathit{push}(e, s)) = s$).

In the sequel we discuss and illustrate five methods, with increasing expressive power, to specify software components. All allow run-time checking, based on the component's execution history.

4.1. INVARIANTS

The first specification is a simple propositional formula built from the constant/observer operations, playing the role of propositional variables.

$$
\begin{aligned}
&\mathit{invariant} ::= \\
&\quad \mathit{constant} \mid \mathit{observer} \mid \\
&\quad \neg\mathit{invariant} \mid \mathit{invariant} \vee \mathit{invariant} \ldots
\end{aligned}
$$

The specification is required to hold invariantly for every state in the execution sequence, represented by the symbol s. Note the difference between constants and observers and how they depend on the state.

$$
\begin{array}{lll}
s \models \mathit{con} & \text{iff} & \mathit{con} \\
s \models \mathit{obs} & \text{iff} & \mathit{obs}(s) \\
s \models \neg\mathit{inv} & \text{iff} & \text{not } s \models \mathit{inv} \\
s \models \mathit{inv}_1 \vee \mathit{inv}_2 & \text{iff} & s \models \mathit{inv}_1 \text{ or } s \models \mathit{inv}_2 \ldots
\end{array}
$$

Consider two example invariants for the stack component: the length of the stack never exceeds the bound and if the last operation was *push* then the value of *top* equals the argument to *push*:

$$
\begin{aligned}
&\mathit{len} \leq \mathit{bnd} \\
&\mathbf{op} = \mathit{push} \ \wedge \ \mathit{top} = \mathbf{arg}
\end{aligned}
$$

The first is the first-order formula, written with a relation over Nat. The second is the higher-order formula where **op** returns the name of the last state-changing operation and **arg** gives the arguments to this operation. The first and higher orders are explained later.

4.2. ACTIONS

Invariants allow us only to formulate properties about individual states, to hold statically over the whole execution. In contrast, an *action* is a propositional formula over pairs of states. It is built from constants and two kinds of observer operations: evaluated in the first state (a pre-state) and in the second state (a post-state), the latter written with a prime. A pre-state is the state before the execution of a generator/modifier operation, a post-state is the state after the execution.

$$
\begin{aligned}
&\mathit{action} ::= \\
&\quad \mathit{constant} \mid \\
&\quad \mathit{observer} \mid \mathit{observer}' \mid \\
&\quad \neg\mathit{action} \mid \mathit{action} \vee \mathit{action} \ldots
\end{aligned}
$$

The action is required to hold over any pair of adjacent states in the execution sequence, here represented by the symbols s_1 and s_2. Note that observers without prime refer to s_1 and observers with prime to s_2; constants refer to none.

$$
\begin{array}{lll}
s_1, s_2 \models \mathit{con} & \text{iff} & \mathit{con} \\
s_1, s_2 \models \mathit{obs} & \text{iff} & \mathit{obs}(s_1) \\
s_1, s_2 \models \mathit{obs}' & \text{iff} & \mathit{obs}(s_2) \\
s_1, s_2 \models \neg \mathit{act} & \text{iff} & \text{not } s1, s2 \models \mathit{act} \\
s_1, s_2 \models \mathit{act}_1 \vee \mathit{act}_2 & \text{iff} & s_1, s_2 \models \mathit{act}_1 \text{ or } s1, s2 \models \mathit{act}_2 \ldots
\end{array}
$$

Consider two example actions for the stack component: a *push* operation increments the length of the stack, provided the stack is not full, and a *pop* operation decrements the length, provided the stack is not empty. Written as higher-order actions:

$$
\begin{array}{l}
\mathit{len} > 0 \Rightarrow \mathbf{op}' = \mathit{pop} \wedge \mathit{len}' = \mathit{len} - 1 \\
\mathit{len} < \mathit{bnd} \Rightarrow \mathbf{op}' = \mathit{push} \wedge \mathit{len}' = \mathit{len} + 1
\end{array}
$$

4.3. PATTERNS

Actions will not suffice for specifying those components which correct behavior in a given state depends not only on the observations in this and the preceding state but also the states before. For instance to check for a stack if the value returned by *top* is correct (if not just inserted) we have to find the most recent state in the history in which the value of *len* equals the current *len* value, then check if the *top* value in this state and the value of *top* in the current state are the same. Here the length of the search depends on the value of an observer.

The idea is to use regular expressions, similar like for specifying words over an alphabet, but the alphabet are vectors of observer values and words are sequences of such records (histories). A pattern is a regular expression build with usual operators $+$ (sum), $\cdot$ (concatenation), * (Kleene's star) and λ (empty sequence), which basic components are invariants and actions.

$$
\begin{array}{l}
\mathit{pattern} ::= \\
\quad \lambda \mid \\
\quad [\mathit{invariant}] \mid [\mathit{action}] \mid \\
\quad \mathit{pattern} + \mathit{pattern} \mid \mathit{pattern} \cdot \mathit{pattern} \mid \mathit{pattern}^*
\end{array}
$$

The operators have the usual interpretation over sequences (histories). We denote such a history by the symbol t and its elements by s, as before.

Special role in the evaluation plays the first element in the history, $hd(t)$ (if one exists), which is used to evaluate all primed observers in action components of the pattern. If t is empty then the only satisfied pattern is λ, otherwise we use the head of t as well as t itself to carry out evaluation $hd(t), t \models pat$.

$$t \models pat \quad \text{iff} \quad \begin{cases} pat = \lambda & \text{if} \quad ln(t) = 0 \\ hd(t), t \models pat & \text{if} \quad ln(t) > 0 \end{cases}$$

In general, evaluation takes the form $s, t \models pat$ where s is a given state and t is a sequence of states, depending on the shape of the pattern pat:

1 $pat = \lambda$
Then t must be empty.

2 $pat = [inv]$
Then t must consist of only one state which satisfies inv, as in Section 4.1. However, this state is not necessarily the most recent state but any state in the history, as determined by the matching process. Also the same invariant may be used a number of times against different states.

3 $pat = [act]$
Then t must consist of only one state, such that s and this state satisfy act, as in Section 4.2. Primed observers always refer to s and unprimed to $hd(t)$. Unlike before, those states are not necessarily adjacent in the history: the post-state is always the most recent state (s), the pre-state is the currently matched state ($hd(t)$). The action can also be used many times against different pairs of states; the same post-state, different pre-states.

4 $pat = pat_1 + pat_2$
Then s and t must either satisfy pat_1 or pat_2, or both.

5 $pat = pat_1 \cdot pat_2$
Then it should be possible to split t into some t_1 and t_2 such that s and t_1 satisfy pat_1 and s and t_2 satisfy pat_2. s remains the same in both cases, providing the same interpretation for the primed observers in the actions.

6 $pat = pat_1^*$
Then either t is empty or can be split into t_1 and t_2 such that s and t_1 satisfy pat_1 and s and t_2 again satisfy pat_1^*. Like above, s is the same in both cases. Note that the same pattern appears at both sides of the definition.

Formally, we define the relation $s, t \models pat$ as follows:

$$
\begin{array}{lll}
s, t \models \lambda & \text{iff} & ln(t) = 0 \\
s, t \models [inv] & \text{iff} & ln(t) = 1 \wedge hd(t) \models inv \\
s, t \models [act] & \text{iff} & ln(t) = 1 \wedge s, hd(t) \models act \\
s, t \models pat_1 + pat_2 & \text{iff} & s, t \models pat_1 \vee s, t \models pat_2 \\
s, t \models pat_1 \cdot pat_2 & \text{iff} & \exists_{t_1, t_2} \quad t = t_1 : t_2 \wedge \\
 & & \qquad s, t_1 \models pat_1 \wedge s, t_2 \models pat_2 \\
s, t \models pat_1^* & \text{iff} & ln(t) = 0 \vee \exists_{t_1, t_2} \quad t = t_1 : t_2 \wedge \\
 & & \qquad\qquad s, t_1 \models pat_1 \wedge s, t_2 \models pat_1^*
\end{array}
$$

We carry out pattern-matching with every modification of the history, i.e. after invocation of every modifier/generator operation, from the most recent state towards the initial state. If we can run successful pattern-matching with every modification of the component's execution history then it means that up till now the component behaved in a proper way. In other words every non-empty suffix of the execution history must satisfy the pattern to regard the component as behaving correctly. As an example consider the pattern below.

$$[a \vee b] + [a \vee b] \cdot [a \Leftrightarrow \neg b]^* \cdot [c' \Leftrightarrow a \vee b]$$

It says the following: either the history contains only one state such that $a \vee b$ holds, or at least two states such that $a \vee b$ holds in the last state, $a \vee b$ in the first state equals c in the last, and $a \Leftrightarrow \neg b$ holds for all intermediate states (if any). Matching this pattern against the history is shown below. We can see that the pattern is satisfied for all non-empty suffixes until state three. However, the value of $a \vee b$ in the first state is different than c in the fourth state, thus the pattern is not satisfied by the history suffix containing the states one to four.

	$[a \vee b] + [a \vee b] \cdot [a \Leftrightarrow \neg b]^* \cdot [c' \Leftrightarrow a \vee b]$							
check					$\times$	$\surd$	$\surd$	$\surd$
states	8	7	6	5	4	3	2	1

a	0	1	1	1	1	0	1	0
b	0	1	0	0	1	1	0	1
c	1	1	0	1	0	1	1	0

4.4. FIRST ORDER PATTERNS

We now demonstrate how to extend such specifications into first- and higher-order patterns. Suppose first-order specifications allow for observers which return integer values. This change will not directly affect patterns but their elements, it is invariants and actions. They are both build from integer expressions involving literals, arithmetic operators, observer names with or without prime, relations between expressions and propositional connectives. Such first order specifications are matched against the history table which contains integer values.

As a simple illustration, consider the first-order extensions to concrete invariants, actions and patterns, as below, and how they are matched against the execution history. The invariant is matched successfully for every state until it fails in state eleven. The action is matched successfully for every pair of states until it fails for states eight and nine. The pattern is matched successfully for every non-empty suffix of the history until and including state five, but it fails in state six.

	$a + b \geq c + 2$										
check	$\times$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$

	$a' + b' \geq c' + 2 \wedge a' > b$										
check			$\times$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	

	$[a + b \geq c + 2]\cdot$ $(\lambda + [a' > b] \cdot (\lambda + [true]^* \cdot [a' + a = b' + b]))$										
check						$\times$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$

a	5	6	5	9	2	4	5	3	2	4	1
b	3	4	3	7	0	1	3	1	0	1	3
c	7	8	5	6	0	1	6	2	0	1	2

4.5. HIGHER ORDER PATTERNS

With higher order patterns we allow observers of any type. Each type comes with its own set of functions and relations. We also assume two special observers: **op** returns the name of the generator/modifier invoked to obtain the current state (initially **op** $= init$) and **arg** returns the set of arguments this generator/modifier was invoked with (if any). From now on, we will also assume a $[true]^*$ trailer for any regular expression used: any prefix of the execution history can be used to satisfy the pattern, rather than the whole history.

As an example consider a higher-order pattern for specifying the stack. We have the usual observers *top* and *len* which come with the definition of the stack and two generic observers **op** and **arg**. **op** returns the name of one of the three possible modifier/generator operations: *init*, *push* or *pop*. **arg** returns the argument used for the invocation, only relevant if **op** $= push$. By $\times$ we represent that the observer value is not available.

$$\begin{aligned}&\mathbf{op} : State \to \{init, push, pop\}\\ &\mathbf{arg} : State \to Elem \cup \{\times\}\end{aligned}$$

The pattern includes three possibilities: if the state was produced by *init* then *len* must be zero; if the state was produced by *push* then *top* is the same as the argument of *push* and *len* in the previous state equals *len* in the current state minus one; if the state was produced by *pop* then *len* in the previous state equals *len* in the current state plus one, and the value of *top* in the most recent state with the same *len* value (in all intermediate states the value of *len* is greater than in the current state) is the same as the current *top* value.

$$\begin{aligned}&[\mathbf{op} = init \wedge len = 0] +\\ &[\mathbf{op} = push \wedge top = \mathbf{arg}] \cdot [len = len' - 1] +\\ &[\mathbf{op} = pop] \cdot [len = len' + 1] \cdot [len > len']^* \cdot [len = len' \wedge top = top']\end{aligned}$$

Below we describe how this pattern can be matched against a concrete history where values put on the stack are real numbers. If the entry contains $\times$ then it means that the value of a given observer was not available, for instance **arg** $= \times$ when the corresponding operation **op** does not take any arguments or $top = \times$ when the stack is empty. The pattern is satisfied until and including state nine but is violated for the state ten: the *top* value is 2.14 and *len* is 1 but the most recent state with the same value of *len* has $top = 1.0$.

check	$\times$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
states	10	9	8	7	6	5	4	3	2	1

	$op{=}pop$	$len{=}len'{+}1$	$len > len'$				$len{=}len' \wedge top{\neq}top'$	true		

op	`pop`	`pop`	`pop`	`push`	`push`	`push`	`pop`	`push`	`push`	`init`
arg	$\times$	$\times$	$\times$	3.5	5.9	0.2	$\times$	2.14	1.0	$\times$
top	2.14	0.2	5.9	3.5	5.9	0.2	1.0	2.14	1.0	$\times$
len	1	2	3	4	3	2	1	2	1	0

5. EXAMPLE: LINE EDITOR

As an illustration of patterns, consider the example of a line editor. The editor provides some basic operations to modify the list of elements depending on the position of a caret. The position can be inside the line and one past the last element (to insert at the end). We have one generator *init* which makes the line empty and sets the caret position at one. Four modifiers: *right* moves the caret forward, if inside the line, *left* moves the caret backward, if not on the first position, *insert* inserts an element at the current position then moves the caret forward, and *delete* deletes an element at the current position, if inside the line. Three observers: *ln* returns the length of the line, *ps* returns the current position of the caret (a natural number) and *ch* returns the element in the line on the current position, if inside the line. Here are the signatures:

type	signatures
$State, Elem$	$init : State$
	$insert : Elem \times State \to State$
	$delete, left, right : State \to State$
	$ln, ps : State \to Nat$
	$ch : State \to Elem$

The pattern defining the behavior of the editor is a sum of five patterns, depending which modifier/generator operation was executed last:

$$pat\text{-}init + pat\text{-}insert + pat\text{-}delete + pat\text{-}right + pat\text{-}left$$

pat-init only requires that the position value becomes one and the length becomes zero in the state resulting from the execution of *init*. The previous state, if any, is not consulted (no primed observers):

$$pat\text{-}init =_{def} [\mathbf{op} = init \wedge ps = 1 \wedge ln = 0]$$

pat-insert requires that the value of the position and length after the execution of *insert* both increase by one, and the character under caret remains the same, but only if the caret is inside the line (we avoid applying the observer *ch* when the caret does not point at any character).

$$pat\text{-}insert =_{def} [\mathbf{op} = insert]\cdot \\ [ln' = ln + 1 \wedge ps' = ps + 1 \wedge (ps \leq ln \Rightarrow ch' = ch)]$$

pat-delete breaks into three cases, depending if the current position, after the execution of *delete*, is greater, equal or less than the length. If greater then we only require that position and length remain unchanged. If equal then the position must be the same and the length must decrease by one, this is when we just deleted the character on the last position in the list. In both cases the resulting position is outside the line, so we are not interested in the value of *ch*. If less then the position must not change and the length must decrease, like before, but we also search the history to find the value corresponding to *ch*. We look for the state where the caret position equals the current one, only counting from the end of the list. This is because prior to *delete*, the caret was on the left of the current character, so the list could be modified. We look for the first state which position has the same distance from the end of the list as the current position, $ln - ps = ln' - ps'$, where we check that the character under caret and the current character are the same.

$$
\begin{array}{l}
\mathit{pat\text{-}delete} =_{def} [\mathbf{op} = \mathit{delete}]\cdot \\
\quad \left(\begin{array}{l}
[ps' > ln' \wedge ps' = ps \wedge ln' = ln]+ \\
[ps' = ln' \wedge ps' = ps \wedge ln' = ln - 1]+ \\
[ps' < ln' \wedge ps' = ps \wedge ln' = ln - 1]\cdot \\
\quad [ln - ps > ln' - ps']^{*}[ln - ps = ln' - ps' \wedge ch' = ch]
\end{array} \right)
\end{array}
$$

pat-right is similar to *pat-delete*. It breaks into three cases for the value of position which is greater, equal or less then the length. Position must increase, provided inside the line, and the length remains the same. We look for the value of the current character like we did for *pat-delete*, counting the distance from the caret position to the end of the line.

$$
\begin{array}{l}
\mathit{pat\text{-}right} =_{def} [\mathbf{op} = \mathit{right}]\cdot \\
\quad \left(\begin{array}{l}
[ps' > ln' \wedge ps' = ps \wedge ln' = ln]+ \\
[ps' = ln' \wedge ps' = ps + 1 \wedge ln' = ln]+ \\
[ps' < ln' \wedge ps' = ps + 1 \wedge ln' = ln]\cdot \\
\quad [ln - ps > ln' - ps']^{*}[ln - ps = ln' - ps' \wedge ch' = ch]
\end{array} \right)
\end{array}
$$

pat-left has also three cases. The first is when the caret is already on the first position, then the length, position and character (provided the caret is inside the line) must stay the same. Otherwise the length remains unchanged and position must decrease, moreover if the previous operation was *insert* then its argument must be the current character. Otherwise we look for: (1) the earliest state with the same position value, which character and the current one must then be the same, or (2) the state where position is one greater than the current and *insert* as the last operation, in which case the argument to *insert* and the current character must be equal. The complication in this case is due to the fact that *insert* not only inserts a character but also forwards the caret.

$$
\textit{pat-left} =_{def} [\mathbf{op} = \textit{left}]\cdot
\left(\begin{array}{l}
[ps' = 1 \wedge ps' = ps \wedge ln' = ln \wedge (ln' > 0 \Rightarrow ch' = ch)]+ \\
[ps' > 1 \wedge \mathbf{op} = \textit{insert} \wedge \\
\quad ps' = ps - 1 \wedge ln' = ln \wedge ch = \mathbf{arg}]+ \\
[ps' > 1 \wedge \mathbf{op} \neq \textit{insert} \wedge ps' = ps - 1 \wedge ln' = ln]\cdot \\
\quad [ps > ps' \wedge (ps = ps' + 1 \Rightarrow \mathbf{op} \neq \textit{insert})]^{*}\cdot \\
\quad [ps = ps' \wedge ch' = ch \vee \\
\qquad ps = ps' + 1 \wedge \mathbf{op} = \textit{insert} \wedge ch' = \mathbf{arg}]
\end{array}\right)
$$

One comment is in place. This pattern may appear complicated, given a rather simple component it is supposed to check. There are two reasons to explain this. The first is the fact that we specified the component in a complete way, when normally one would like to specify and check selected critical properties. The second is the fact that this pattern has been defined directly in terms of observers, without the intermediate level of auxiliary properties, specially designed to capture reoccuring properties. On the other hand, the pattern above is composed of five independent patterns which, one can argue, are simple on their own. Whatever point of view we adopt, the scalability of the whole approach is no doubt an important practical concern, which we plan to address in our future work, along with other issues discussed in the conclusions.

6. IMPLEMENTING THE WRAPPER-GENERATOR

One way to make sure that a given component is fail-stop is to prove that it satisfies the fail-stop specification. But this ignores the special form of the fail-stop property, and typically requires a fair amount of human assistance. Instead, we want to guarantee this property at run-time, where a specification is given in the form of a pattern and run-time checking is carried out by an automatically generated wrapper. Here we discuss the implementation of the wrapper-generator.

Consider the structure of the wrapped component. The signature is the same as that of the original component plus the observer *error*. The state includes some part of the execution history (as necessary for checking) and the error indicator. The wrapper takes over all communication between the original component and its users. Invocation of a constant/observer is passed to the component and obtained results directly returned to the user, no checking is done in this case. Invocation of a generator/modifier is carried out on the original component, observations about the new state and operation itself are recorded. Then we carry out pattern-matching with respect to this modified history and set the error flag accordingly. The wrapper may conduct more activities, in particular maintain the history record, remove the observations with no effect on future checking etc. Figure 2 depicts this structure.

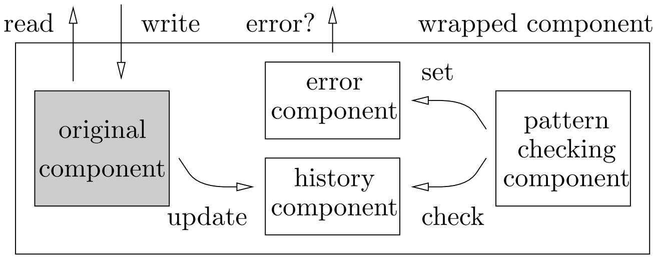

Figure 2 Structure of the wrapped component.

We would like this wrapped component to be generated automatically for a given pattern. We designed a prototype wrapper generator for components written in a small subset of Java. The generator is intended as a tool for writing case studies on the use of patterns for specifying components and pattern-matching for checking their behavior at run-time. At this moment there are several constraints we impose on the input Java classes: the only data type is natural numbers, modifiers take zero or one argument, class constructors take no arguments and exceptions are not allowed. The wrapper generator $Wrap$ is itself written in Java, using the JavaCC tool to support parsing. The input is the specification file that contains the name of the class to be wrapped, its signature in a simplified form and the pattern specifying the correct behavior of the class. Here is the specification file for the stack:

```
#Classname   | #Actions                          | #Pattern
 NatStack    | q0 = {op = init /\ len = 0}       | q0 +
#Modifiers   | q1 = {op = push /\ arg = top'}    | q1.q2 +
 push();     | q2 = {len = len' - 1}             | q3.q4.q5*.q6
 pop;        | q3 = {op = pop}                   |
#Observers   | q4 = {len = len' + 1}             |
 top;        | q5 = {len > len'}                 |
 len;        | q6 = {len = len' /\ top = top'}   |
```

$Wrap$ produces the source code for the wrapper class, which is a subclass of the class specified in the input file. The class contains some additional variables to store the execution history, the automaton which represents the regular expression, error indicator, methods to update and check the execution history etc. All observers are inherited from the parent class and all modifiers and generators are overridden: call the parent method, update the history record, carry out pattern-matching and set accordingly the error indicator. *Wrap* also creates an applet which can be used to test the execution of the produced wrapped class. The applet allows to invoke modifiers, individually or in a sequence, display the execution history, observer values and indication of an error. It also compares the execution times for original and the wrapped class.

A few comments on the implementation and the use of wrappers. As our main purpose is error-detection, we do not consider how the wrapped component should be used to build reliable distributed systems. This belongs to the next level – implementation of fault-tolerance. One idea would be to integrate patterns with a component framework like CORBA: make them part of the IDL description of a component, generate the code for run-time checking along with the usual stub code, build applications which actively inspect the error status for the components they are built from. Another idea is to use the wrapper in a remote way, as a smart proxy for its component, or as a CORBA interceptor to become part of the growing class of objects providing infrastructure services for other objects. Whatever method is used, wrappers represent the knowledge (reflection) how their components should behave. They cannot change how the components behave, but the behavior of applications built from such components.

7. CONCLUSIONS

We demonstrated how regular expressions can be used to formally specify software components, in order to be able to check their behavior at run-time. We build such expressions from the propositions about the pairs of states of a component, one of which is the current state, another is some previous state determined by the checking process. Checking is carried out by a specification-generated wrapper, as a kind of pattern-matching, which produces a fail-stop component [14] from a component which may fail in an arbitrary way. We also presented an architecture of the wrapped component and the prototype wrapper-generator, for components written as Java classes.

We argue that run-time checking is particularly suitable for open, object-based distributed systems. Distribution makes testing such systems difficult in general, given the large number of components and many possible ways for them to interact. Openness means such systems are relatively easy to modify, but also hard to verify. In particular, deciding at run-time which components should be used (dynamic invocation) makes it hard to approach an apriori verification. Openness also means the components come from many origins, they may lack proper certification or make it difficult (being remote or proprietary) to inspect the quality ourselves. Moreover, decisions to include a component are based on its IDL description, which typically lacks the semantic information or describes this semantics in a natural language. Run-time checks provide often the only method of protection for the whole system from its unsound, unreliable components. The paper showed how to introduce such checks in a systematic way, generated from formal specifications.

This paper is a revised version of [6]. Related work on software specification with regular expressions include [2], but the focus is symbolic reasoning, not run-time checking. Specification-based testing [12] is another related area which purpose is mainly analysis (off-line), unlike here where we try to improve reliability (on-line). Run-time checks are practically implemented in the Java Assert class (of java.lang.object) but only to check invariants. One more related area is fault-tolerance [7]. Formalization of fault-tolerance has been carried out by several authors, e.g. [8, 5], where they explain how to formally specify and verify an existing system. Here, in contrast, we provide a constructive approach to actually build such systems, the wrapper-generator, although for now we focus only on error-detection.

We plan to continue this work in several directions. First, we plan to produce some real-life case studies for specifying components with patterns. Among others we look at the components described in the CORBA services documents. Based on such case studies we intend to study how the approach scales up. Another direction is to further investigate the foundations for run-time checking: we continue to look at co-algebras, with their approach to specification of classes [4], but also at the alternating automata [1]. It remains to see how we can relate patterns to the more abstract algebraic or co-algebraic specifications, and how to define refinement between patterns according to their strength. We also plan to study how run-time checking with pattern-matching can be used to build fault-tolerant systems, in particular how to verify fault-tolerance on the system level [5] based on the run-time checking on the component level. This could provide a demonstration how symbolic and run-time techniques can work together. Then there are real-time and memory requirements for the whole scheme to work in practice. It must be possible to analyze patterns to see which parts of the history we have to remember for future checking and which we can discard. At best we would like to analyze patterns statically in this respect, at worst include some run-time mechanisms like garbage collection. We also want to analyze patterns with respect to the performance overheads.

Acknowledgments

We wish to thank anonymous referees for their useful, constructive comments.

References

[1] A.K. Chandra and L.J Stockmeyer. Alternation. In *FOCS 17*, pages 98–108, 1976.

[2] M. Dwyer, G. Avrunin, and J. Corbett. Patterns in property specifications for finite-state verification. In *International Conference on Software Engineering*, 1999.

[3] B. Jacobs and J. Rutten. A tutorial on (co)algebras and (co)induction. *EATCS Bulletin*, 62:222–259, 1997.

[4] Bart Jacobs. Objects and classes, co-algebraically. In *Object-Orientation with Parallelism and Persistence*. Kluwer, 1996.

[5] T. Janowski. On bisimulation, fault-monotonicity and provable fault-tolerance. In *Proc. 6th AMAST*, volume 1349 of *LNCS*, 1997.

[6] T. Janowski and W. Mostowski. Fail-stop software components by pattern matching. Presented at the Workshop on Run-Time Result Verification, part of the Federated Logic Conference, Trento, 1999.

[7] J.C. Laprie. Dependability: Basic concepts and associated terminology. Technical report, PDCS, 1990.

[8] Z. Liu and M. Joseph. Specification and verification of fault-tolerance, timing, and scheduling. *ACM TOPLAS*, 21(1), 1999.

[9] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[10] C. Pfister and C. Szyperski. Why objects are not enough. In *Int. Component Users Conference, Munich, Germany*, 1996.

[11] The RAISE Language Group. *The RAISE Specification Language*. Prentice Hall, 1992.

[12] D. Richardson, O. O'Malley, and C. Tittle. Approaches to specification-based testing. In *ACM Symposium on Software Testing, Analysis and Verification*, 1989.

[13] Donald Sannella and Andrzej Tarlecki. Essential Concepts of Algebraic Specification and Program Development. *Formal Aspects of Computing*, 9:229–269, 1997.

[14] R.D. Schlichting and F.B. Schneider. Fail stop processors: An approach to designing fault-tolerant computing systems. *ACM TOCS*, 1(3):222–238, 1983.