Dynamic aspects of retrenchments through temporal logic

Richard Banach[1], Jean-Paul Bodeveix[2], Mamoun Filali[2], and Michael Poppleton[3]

[1] Department of Computer Science, University of Manchester
Manchester M13 9PL, UK
Email: banach@cs.man.ac.uk
[2] IRIT Université Paul Sabatier
118 route de Narbonne, F-31062 Toulouse Cedex, France
Email: {bodeveix,filali}@irit.fr
[3] Department of Electronics and Computer Science
University of Southampton, Highfield
Southampton SO17 1BJ, UK
Email: mrp@ecs.soton.ac.uk

Abstract. Refinement is used as a way to verify an implementation with respect to a specification. States of related systems are linked through a so called gluing invariant which remains always true during the synchronous execution of both systems. Refinement is a sufficient condition for this property. Retrenchment is a generalization of refinement which relax the constraints between both systems. This paper propose a temporal logic counterpart for some specific forms retrenchment.

1 Introduction

Usually, the correctness of an implementation with respect to a specification is defined by the inclusion between the set of concrete traces allowed by the behaviors of the implementation and the set of abstract traces allowed by the abstract specification. Refinement [3, 6] is a sufficient condition for trace inclusion. It is specified using a *gluing invariant* relating the state space of abstract and concrete machines. Refinement establishes by induction that the gluing invariant is always true. With respect to temporal logic, such a property can be seen as the expression of a strict synchronization between the abstract specification and the concrete implementation. If we ignore stuttering steps, the abstract specification and the concrete implementation appear to be working in a lock step way.

Retrenchment [4] is a weakening of refinement that first allows input/output conversions and second does not enforce a strict synchronization between the abstract specification and the concrete implementation. Actually, it does not require the preservation of the gluing invariant at all. In general, no temporal counterpart of the always nontrivially valid property associated to refinements exists for retrenchments.

In this study, we show how temporal properties can be used to specify how the synchronization between the abstract specification and the concrete implementation can be lost. We also elaborate a temporal proof process using retrenchments similar to the one using refinements. After proposing a temporal relation that should exist between a specification and a "retrenched" implementation, we show that retrenchment is a sufficient condition to establish the correctness of a "retrenched" implementation with respect to it.

This paper is organized as follows. In Section 2, we recall the notions of labelled transition systems and their specifications in TLA. Section 3 introduces the notion of retrenchments over operations and over traces. Section 4 illustrates retrenchments through some examples. Section 5 elaborates some trace properties of retrenchments. Section 6 concludes.

2 Transition systems and refinements in TLA

Retrenchment is defined as a variation of refinement of B machines. In order to define the semantics of refinement and retrenchment as well as temporal logic properties of execution traces, we need a lower level formalism and we have chosen the TLA specification language for this purpose. As B, it introduces variables and before-after relations but does not provide any built-in notion of refinement. However, it offers linear time temporal operators allowing the expression of trace properties and supports the definition of refinement or retrenchment. This section first presents TLA then how TLA is used to specifiy transition systems.

2.1 A brief introduction to TLA

TLA+ specifications [7] are organized into modules. A module contains constants, variables, assumptions and definitions. TLA+ defines basic set constructors. We will mainly use the following ones:

- opaque set constructors define sets without knowledge of their elements.
- $[D \rightarrow R]$ is the set of functions from D to R.
- $[f_1 : S_1, \ldots, f_n : S_n]$ is the set of records whose fields $f_{i\,\in\,1..n}$ are elements of the sets $S_{i\,\in\,1..n}$ respectively.

We are concerned with transition systems [2]. While their state spaces can be defined using variables with values in sets as just given, TLA+ definitions are used to introduce the following:

- The set of initial states, using a predicate usually called `Init`.
- The set of transitions, using action predicates. An action is a formula containing primed (next state) variables and unprimed (current state) variables. Such a formula describes the relation between the current state and next state values of the variables.

  For refinement and composition purposes, TLA introduces the notations $[A]_v$ for $A \vee v' = v$ and $\langle A \rangle_v$ for $A \wedge v' \neq v$ where v is a state function.

$[A]_v$ expresses that either an A step or a stuttering step with respect to v occurred, while $\langle A \rangle_v$ expresses that an A step actually occurred. The global transition relation is usually introduced as the `Next` action predicate.
– The dynamics of a transition system is specified through temporal operators. Safety properties are specified through the $\Box$(always) operator of temporal logic. In the same way, liveness properties are specified through the $\Diamond$(eventually) operator of temporal logic. For liveness properties, the $\leadsto$(leadsto) operator is also usually used. $A \leadsto B$ is defined as $\Box(A \Rightarrow \Diamond B)$.

A TLA+ module can be considered to be parameterized by its constants and variables. A module can be instantiated by setting these, for example:

$$\text{anInstance}(e_1, \ldots, e_n) == \textbf{INSTANCE } \text{module } \textbf{WITH } v_1 \leftarrow e_1, \ldots, v_n \leftarrow e_n$$

where $e_1, \ldots, e_n$ are expressions which will replace the respective occurrences of formal parameters $v_1, \ldots, v_n$ in the instance.

In TLA, we can hide a variable with the existential quantifier $\exists\!\!\exists$ of temporal logic. The formula $\exists\!\!\exists$ v: F means that there exists a sequence of values that can be assigned to the variable v that will make the formula F true.

Last but not least, in a TLA+ module we can state theorems.

2.2 Transition systems in TLA

A transition system is a tuple $\langle Q, I, R \rangle$ where Q is a set, $I \subseteq Q$ is the set of initial states and $R \subseteq Q \times Q$ is the transition relation. In order to be more usable in practice, the state space is often modeled by a function over a set of state variables. Then I and R are respectively specified by a predicate over the state variables and by a predicate over two copies of the state variables denoting before and after states. TLA supports the direct encoding of such a transition system (see module `trs`).

MODULE *trs*

CONSTANTS S

$$
\begin{aligned}
&\mathit{isSys} \triangleq \\
&\quad \land\ S = [\mathit{data} \mapsto S.\mathit{data},\ \mathit{init} \mapsto S.\mathit{init},\ \mathit{next} \mapsto S.\mathit{next}] \quad \text{shape of the record} \\
&\quad \land\ S.\mathit{init} \in [S.\mathit{data} \rightarrow \text{BOOLEAN}\ ] \quad \text{predicate characterizing initial states} \\
&\quad \land\ S.\mathit{next} \in [S.\mathit{data} \times S.\mathit{data} \rightarrow \text{BOOLEAN}\ ] \quad \text{transition relation}
\end{aligned}
$$

ASSUME *isSys*

VARIABLES u

$\mathit{Init} \triangleq u \in S.\mathit{data} \land S.\mathit{init}[u]$

$\mathit{Next} \triangleq S.\mathit{next}[\langle u,\ u' \rangle]$

$\mathit{Spec} \triangleq \mathit{Init} \land \Box[\mathit{Next}]_u$

The behavior of the system, corresponding to its set of traces, is specified by the temporal formula `Spec` which has, in general, the following form:

$$\text{Spec} \triangleq \text{Init} \wedge \Box[\text{Next}]_{\text{vars}} \wedge \text{Liveness}$$

This formula defines a property about the sequence of values taken by the state variables. The initial value satisfies the `Init` predicate and the `Next` predicate is satisfied by consecutive values. The `vars` index allows stuttering: consecutive values may be identical arbitrarily many times, which leaves holes where other parts of the system can evolve. Infinite stuttering can be avoided, for example by stating a liveness property.

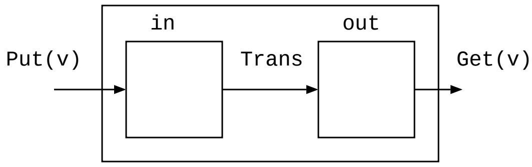

Fig. 1. A transmitter in TLA

Example: A transmitter in TLA A one step transmitter (see fig. 1) can be descibed by a TLA module communicating through the two shared variables `in` and `out`. `in` equals `empty` when the module is ready to consume a new value. `out` is not empty when the module is ready to produce a new value. Three operations are provided:

- `Put(v)` saves the value `v` in the variable `in` if it is empty.
- `Trans` transfers a non empty value from `in` to `out`.
- `Get(v)` waits for the value `v` to be in the variable `out`.

The transition relation of the system, corresponding to the `Next` predicate, is defined as the disjunction of these, the elementary transitions.

MODULE *transmitter*

CONSTANTS $Data$, $empty$

ASSUME $empty \notin Data$

VARIABLES in, out

$$TypeInvariant \triangleq in \in Data \cup \{empty\} \wedge out \in Data \cup \{empty\}$$

$$Init \triangleq in = empty \wedge out = empty$$

$$Put(v) \triangleq v \in Data \wedge in = empty \wedge in' = v \wedge \text{UNCHANGED } out$$

$$Trans \triangleq in \neq empty \wedge out = empty \wedge out' = in \wedge in' = empty$$

$$Get(v) \triangleq out \neq empty \wedge v = out \wedge out' = empty \wedge \text{UNCHANGED } in$$

$$Next \triangleq (\exists\, v : Put(v)) \vee (\exists\, v : Get(v)) \vee Trans$$

$$Spec \triangleq Init \wedge \Box[Next]_{\langle in,\, out\rangle}$$

This system can also be seen as an instance of the meta level of the notion of a transition system. Due to the lack of space, in the following, the examples will only be specified at the object level.

2.3 Input-Output transition systems in TLA

Retrenchment was introduced in the B framework, in which a transition is labelled by an operation name and its input and output parameters. We therefore split the label into an input and an output label, the input label modelling both the operation name and its input arguments. Consequently, the state space contains now three variables: i, o and u. The signature of the `next` predicate is modified accordingly. Note that stuttering is here introduced through the `none` operation name which leaves the state unchanged. It has to be noted that Input-Output transitions systems defined here differ from I/O automata [8] where both input and output are events to which the automaton reacts by updating its internal state. The direction of the event in I/O automata is only important for composition purposes.

MODULE *iotrs*

CONSTANTS $Input, Output, S, none$

VARIABLES i, u, o

$$
\begin{aligned}
& isTrs \triangleq \\
& \quad \land S = [data \mapsto S.data,\ init \mapsto S.init,\ next \mapsto S.next] \\
& \quad \land S.init \in [S.data \to \text{BOOLEAN}] \\
& \quad \land S.next \in [Input \cup \{none\} \times S.data \times S.data \times Output \cup \{none\} \to \text{BOOLEAN}]
\end{aligned}
$$

ASSUME $isTrs \land none \notin Input \land none \notin Output$

$$Init \triangleq S.init[u] \land i \in Input \cup \{none\} \land o \in Output \cup \{none\}$$

$$
\begin{aligned}
& Next \triangleq \\
& \quad \land i \in Input \cup \{none\} \land i' \in Input \cup \{none\} \\
& \quad \land o \in Output \cup \{none\} \land o' \in Output \cup \{none\} \\
& \quad \land S.next[i, u, u', o]
\end{aligned}
$$

$$Spec \triangleq Init \land \Box(Next \lor (i = none \land o = none \land \text{UNCHANGED } u))$$

Remark The conjunct $i' \in Input \cup \{none\}$ allows for *composition* with another component that will provide the next input.

2.4 Refinements

In the sequel we will use the following notations:

- op_a,(resp. op_c) denotes an abstract (resp. concrete) operation,
- u, u',(resp. v, v') denote the before and after states associated to an abstract, (resp. concrete) operation.
- i, (resp. j) is the input of the abstract (resp. concrete) operation.
- o, (resp. p) is the output of the abstract (resp. concrete) operation.

These notations are summarized in the following table.

	Transition	*Before*	*After*	*Input*	*Output*
Abstract	op_a	u	u'	i	o
Concrete	op_c	v	v'	j	p

Definition 1 (Operation refinement) *The abstract operation op_a is said to be refined by the concrete operation op_c through the gluing invariant G if op_a can simulate op_c starting from a concrete state satisfying the gluing invariant G and leading to a state where the gluing invariant is preserved.*

$$\begin{gathered} op_a \sqsubseteq_G op_c \\ \equiv \\ G(u,v) \wedge i = j \wedge op_c(j,v,v',p) \Rightarrow \exists u', o : op_a(i,u,u',o) \wedge G(u',v') \wedge o = p \end{gathered}$$

Definition 2 (trace of a transition system) *Given a transition system S defined as a pair $\langle init, next \rangle$ where init is its initialization predicate and next its transition relation, a trace of S is a sequence of inputs-outputs (i,o) satisfying the predicate $\overline{S}$ where*

$$\overline{S}(i,o) \equiv \boldsymbol{\exists}\, u : S.init(u) \wedge \Box S.next(i,u,u',o)$$

Definition 3 (Trace refinement) *There is a trace refinement between an abstract system S_a and a concrete system S_c, if each trace of the concrete system is a trace of the abstract system. Formally, we define trace refinement as follows*

$$S_a \sqsubseteq S_c \equiv \boldsymbol{\forall}\, i,o : \overline{S_c}(i,o) \Rightarrow \overline{S_a}(i,o)$$

Theorem 1 (Sufficient condition for refinement) *Operation refinement is a sufficient condition for trace refinement:*

$$(\forall v : S_c.init(v) \Rightarrow \exists u : G(u,v) \wedge S_a.init(u)) \wedge S_a.next \sqsubseteq_G S_c.next \Rightarrow S_a \sqsubseteq S_c$$

3 Retrenchments

Retrenchment [5] was introduced as a weakening of B [1] refinement: input and output parameters of an operation and its retrenchment are not necessarily equal and the *retrieve* relation (called the gluing invariant in the context of refinement) is not necessarily preserved. In the latter case, the *concedes* relation must be satisfied. Retrenchment is a means to define a new behavior as a variant of an existing one while enlightening the differences between both specifications. For example, a bounded queue does not refine an unbounded queue except while it does not overflow. Retrenchment has mainly been used for numerical applications where retrenchement computes an approximation of the exact solution specified by the retrenched machine. In these situations, the gluing invariant is not preserved because of imprecise computations. The difference between both computations is specified by the concedes relation.

This section introduces retrenchment machines as an extension of the B language. Retrenchment being introduced at the operation level, we show how it can be expressed at the transition system level without loss of generality, operations being merged into a single relation. Then, we propose a first result relating input/output traces and retrenchment.

3.1 Retrenchment machines

The following gives a skeleton of a B machine and its retrenchment. Note that the `output` and `concedes` relations are defined in the context of the after-state, the before-state being mentioned using the $0 suffix.

```
MACHINE M
  VARIABLES U
  OPERATIONS O ⟵ op(I) ≜ ...
END

RETRENCHMENT R RETRENCHES M
  VARIABLES V
  RETRIEVES G(U,V)
  OPERATIONS P ⟵ op(J) ≜
    BEGIN
    ...
      WITHIN W_op(I,J,U,V)
      OUTPUT O_op(I,J,U,V,U$0,V$0,O,P)
      CONCEDES C_op(I,J,U,V,U$0,V$0,O,P)
    END
END
```

The machine `M` is retrenched (via the RETRENCHES M clause and retrieve relation `G(U, V)`), to machine `R`. The operations of the latter machine now have bodies which are substitutions augmented with the `WITHIN`, `OUTPUT` and `CONCEDES` clauses. The following definition presents the role of each clause and also gives the proof obligation associated to each operation of a retrenched machine. The latter is to be considered as the semantics of retrenchment.

Definition 4 (Retrenchment) *Retrenchment is defined with respect to:*

- *a gluing invariant G,*
- *an Input relation W, playing the role of a precondition with possible input translation.*
- *an Output relation O, enforcing the post gluing invariant and performing output translation.*
- *a Concedes relation C, weakening the post gluing invariant.*

The retrenchment between Op_c and Op_a is formally defined as follows:

$$\begin{aligned} Op_c \ {}^{W}_{C}\!\underset{\sim}{\sqsubset}{}^{O}_{G}\ Op_a \equiv\ & G(u,v) \wedge W(i,j,u,v) \wedge Op_c(v,v',j,p) \\ & \Rightarrow \exists u', o : Op_a(u,u',i,o) \wedge \\ & \quad ((G(u',v') \wedge O(i,j,u,v,u',v',o,p)) \vee C(i,j,u,v,u',v',o,p)) \end{aligned}$$

Remark Defining default values for W, O and C as $i = j$, $o = p$, **FALSE**, respectively, it follows that $\sqsubseteq_G$ reduces to operation refinement.

3.2 Operation retrenchment and transition retrenchment

Retrenchment has been defined in a per operation basis but can be equivalently defined on the global relation `Next`. For this purpose, operation names are represented by a new input argument of the `Next` operation. Operation dependent information such as the within, `output` and `concedes` relations already take the input parameters of the current operation as an argument. A global definition of these relations can thus be given:

$$\texttt{W}(\langle\texttt{op,i}\rangle\texttt{,j,u,v}) = \texttt{W}_{op}\texttt{(i,j,u,v)}$$
$$\texttt{O}(\langle\texttt{op,i}\rangle\texttt{,j,u,v,u0,v0,p}) = \texttt{O}_{op}\texttt{(i,j,u,v,u0,v0,p)}$$
$$\texttt{C}(\langle\texttt{op,i}\rangle\texttt{,j,u,v,u0,v0,p}) = \texttt{C}_{op}\texttt{(i,j,u,v,u0,v0,p)}$$

```
Next_a(op,u,u',i,o) = op_a(u,u',i,o)
Next_c(op,v,v',j,p) = op_c(v,v',j,p)
```

The proof obligation associated to the retrenchment between concrete and abstract `Next` relations is the conjunction of the proof obligations associated to individual operations:

$$\begin{array}{l}
\bigwedge_{op} \{\ G(u,v) \wedge W_{op}(i,j,u,v) \wedge op_c(v,v',j,p) \\
\Rightarrow \exists u', o : op_a(u,u',i,o) \wedge \\
\quad ((G(u',v') \wedge O_{op}(i,j,u,v,u',v',o,p)) \vee C_{op}(i,j,u,v,u',v',o,p))\ \} \\
\equiv \\
\forall\ op : G(u,v) \wedge W(\langle op,i\rangle,j,u,v) \wedge \mathrm{Next}_c(op,v,v',j,p) \\
\Rightarrow \exists u', o : \mathrm{Next}_a(op,u,u',i,o) \wedge \\
\quad ((G(u',v') \wedge O(\langle op,i\rangle,j,u,v,u',v',o,p)) \vee C(\langle op,i\rangle,j,u,v,u',v',o,p))
\end{array}$$

Consequently, the operationwise retrenchment of a machine can be represented by a single retrenchment between global abstract and concrete transition relations, as is usual in TLA. This property justifies, a posteriori, that the operation name can be handled in the same way as an input parameter.

3.3 Trace retrenchment

Retrenchment, as well as refinement, relates the state of two machines, a concrete one and an abstract one. As previously stated, refinement is a sufficient condition for inclusion between the set of traces of two machines. This sufficient condition gives an induction based proof tactic to establish the inclusion and relies on the internal state of both machines.

We now try to express a similar result for retrenchment. Trace retrenchment is defined on sequences of inputs and outputs, while operation retrenchment depends on the internal states of machines.

Definition 5 (Trace retrenchment) *S_c is a trace retrenchment of S_a iff there exist trace retrenchment relations W_t, O_t, C_t such that:*

$$\forall\, i, j, p : \overline{S_c}(j,p) \wedge W_t(i,j) \Rightarrow \exists\, O_t : (\overline{S_a}(i,o) \wedge O_t(i,j,o,p)) \ \mathbf{W}\ (\neg W_t(i,j) \vee C_t(i,j,o,p))$$

*where φ **W** ψ is the weak until operator defined as $\Box\varphi \vee (\varphi\ \mathbf{U}\ \psi)$.*[4]

As for operation retrenchment, trace retrenchment reduces to trace refinement when default values are taken for W_t, O_t and C_t.

Theorem 2 *Operation retrenchment is a sufficient condition for trace retrenchment.*

For this, operation retrenchment is used to build the output sequence step by step while G and W are preserved. The trace retrenchment relations W_t, O_t, C_t are defined as:

$$\begin{array}{ll} W_t(i,j) & = \forall u, v : W(i,j,u,v) \\ O_t(i,j,o,p) & = \exists u, v, u', v' : O(i,j,u,v,u',v',o,p) \\ C_t(i,j,o,p) & = \exists u, v, u', v' : C(i,j,u,v,u',v',o,p) \end{array}$$

Remark In the following, we consider the internal state as hidden. The system is thus seen as an input/output automaton characterized by input/output trace properties expressed in temporal logic. Thus, retrenchment will be seen as a proof method for such properties.

4 Example of retrenchment in TLA

We illustrate the use of retrenchment through the BAG example. Since, we will reason on operations, first we introduce them and then we define the state machines that explicitly use them. In order to illustrate refinement and retrenchment, we consider an abstract specification of bags and a concrete one based on sequences.

4.1 The relational operations

The relational operations manage states explicitly through the parameters `St1` and `St2` containing the before and after states. Additional parameters encode the operations input and/or output arguments. We consider three operations on bags in order to illustrate the management of several types of input/output parameters. `put` and `get` take an element as input and as output. `diff` takes a bag as input, returns the bag containing the difference of the current and input bags, and saves the parameter to the current state.

[4] The weak until operator **W** is not part of TLA.

MODULE *abs_ops*

$\textsc{local instance}\ Bags$

$\textsc{constants}\ Elem$

$init(St) \triangleq St = EmptyBag$

$TypeInvariant(St) \triangleq IsABag(St) \wedge \textsc{domain}\ St \subseteq Elem$

$put(i, St1, St2) \triangleq i \in Elem \wedge St2 = St1 + SetToBag(\{i\})$

$get(St1, St2, o) \triangleq BagIn(o, St1) \wedge St2 = St1 - SetToBag(\{o\})$

$min(a, b) \triangleq \textsc{if}\ a > b\ \textsc{then}\ b\ \textsc{else}\ a$

$inter(b1, b2) \triangleq$
$\quad [e \in (\textsc{domain}\ b1) \cap (\textsc{domain}\ b2) \mapsto min(CopiesIn(e, b1), CopiesIn(e, b2))]$

$diff(i, St1, St2, o) \triangleq St2 = inter(St1, i) \wedge o = St1 - i$

The `conc_ops` module defines the same operations but the state is now a bounded sequence of size at most 10. Input and output parameters of type `Elem` keep the same signature, but bag parameters of the `diff` operation are now typed as sequences. The code for the `diff` operation is not given.

In order to manage the finiteness of the sequence, the `put` operation may overflow. For this purpose, the internal state can be denoted either by a sequence or by the `Oflow` constant. The value returned by `get` is unspecified if the stack has overflowed in the past.

MODULE *conc_ops*

$\textsc{local instance}\ Sequences$

$\textsc{local instance}\ Bags$

$\textsc{constants}\ Elem, Oflow$

$SeqToBag(s) \triangleq$
$\quad \textsc{let}\ F[S \in Seq(Elem)] \triangleq$
$\quad \textsc{if}\ S = \langle\rangle\ \textsc{then}\ EmptyBag\ \textsc{else}\ SetToBag(\{Head(S)\}) + F[Tail(S)]\textsc{in}$
$\quad F[s]$

$init(St) \triangleq St = \langle\rangle$

$put(e, St1, St2) \triangleq$
$\vee\ Len(St1) \leq 9 \wedge St2 = Append(St1, e)$
$\vee\ Len(St1) = 10 \wedge St2 = Oflow$
$\vee\ St1 = Oflow \wedge St2 = St1$

$get(St1, St2, e) \triangleq$
$\vee\ \exists\, r \in Seq(Nat) : St1 = \langle e\rangle \circ r \wedge St2 = r$
$\vee\ St1 = Oflow \wedge St2 = St1$

$diff(i, St1, St2, o) \triangleq \textsc{true}$

4.2 The state machines

The state machine use three state variables corresponding to the inputs, the outputs and the internal state of the machine. The abstract and concrete behaviors are defined using the relational operators. The `Next` transition applies one of the previously defined operators. As usual, the specification of the behaviors of the machine is defined using the temporal operator $\Box$ stating that consecutive values of legal traces satisfy the `Next` relation.

The abstract machine The abstract machine use state variables `i, U, o` to represent its inputs, its internal state and its outputs.

MODULE *abs*

EXTENDS *abs_ops*
LOCAL INSTANCE *Bags*
VARIABLES $i,\ U,\ o$

$Init \triangleq init(U)$

$Put \triangleq put(i,\ U,\ U') \land i' = i \land o' = o$

$Get \triangleq get(U,\ U',\ o') \land i' = i$

$Diff = diff(i,\ U,\ U',\ o') \land i' = i$

$Next \triangleq Put \lor Get \lor Diff$

$Spec \triangleq Init \land \Box[Next]_{\langle i,\ U,\ o\rangle}$

The concrete machine The abstract machine uses state variables `j, V, p` to represent its inputs, its internal state and its outputs.

MODULE *conc*

EXTENDS *conc_ops*, *Sequences*, *Naturals*

VARIABLES $j,\ V,\ p$

$TypeInvariant(W) \triangleq W \in \{v \in Seq(Elem) : Len(v) \leq 10\} \cup \{Oflow\}$

$Init \triangleq init(V)$

$Put \triangleq put(j,\ V,\ V') \land p' = p$

$Get \triangleq get(V,\ V',\ p') \land j' = j$

$Diff \triangleq diff(i,\ V,\ V',\ o') \land i' = i$

$Reset \triangleq V = Oflow \land V' = \langle\rangle$

$Next \triangleq Get \lor Put \lor Diff$

$Spec \triangleq Init \land \Box[Next]_{\langle j,\ V,\ p\rangle}$

Refinement is not sufficient to relate the behaviors of both machines for several reasons:

– the signature of operations has changed: the `Diff` operation exports the representation of the buffer.
– the value returned by the concrete `get` operation is unspecified in case of overflow.

The substitution property allowed by refinement is lost because of the interface change but performing input/output conversion is enough to keep the synchronisation between concrete and abstract executions. A gluing invariant could be that abstract and concrete buffers have the same contents when overflow is false. Now, if the overflow flag is not memorized, a non trivial gluing invariant could be that the contents of concrete and abstract buffers are the same if size of the buffer is less than 10. This property is not preserved by the `get` operation. The *concedes* relation of general retrenchment must be introduced.

5 Trace properties and retrenchments

The next paragraphs present several instances of the general definition of retrenchment that are close to refinement. Each time, we show how the considered instance of the retrenchment definition can be seen as a sufficient condition for some kind of trace refinement. It is then possible to associate temporal properties to retrenchment.

5.1 Refinement with input/output conversion

The general definition of retrenchment can be instantiated so that a one to one correspondence is preserved between abstract and concrete steps, but input and output may change. The correspondence between abstract and concrete input or output is specified by the relations `I` and `O`. They are weaker than for general retrenchment: they only depend on input or output values and do not refer to the abstract or concrete states.

Definition 6 (Operation refinement with input/output conversion)

$$\begin{aligned} op_a\,{}^{I}\!\underset{\sim}{\sqsubset}{}^{O}_{G}\, op_c \equiv\ & G(u,v) \wedge I(i,j) \wedge op_c(v,v',j,p) \\ & \Rightarrow \exists u', o : op_a(u,u',i,o) \wedge G(u',v') \wedge O(i,j,o,p) \end{aligned}$$

The two predicates `I` and `O` can be used to define a trace level notion of refinement with input/output conversion which extends the usual notion of trace refinement.

Definition 7 (Trace refinement with input/output conversion)

$$S_a\,{}^{I}\!\underset{\sim}{\sqsubset}{}^{O}_{*}\, S_c \equiv \forall\, i,j,p : \overline{S_c}(j,p) \wedge (\Box I(i,j)) \Rightarrow \exists\, o : \overline{S_a}(i,o) \wedge \Box O(i,j,o,p)$$

The following theorem states that operation refinement with input/output conversion is a sufficient condition for trace refinement with input/output conversion. It simply extends the usual result.

Theorem 3 (Sufficient condition for refinement)

$$(\forall v : S_c.init(v) \Rightarrow \exists u : G(u,v) \wedge S_a.init(u)) \wedge S_a.next \ {}^{I}\!\underset{\sim}{\sqsubset}{}^{O}_{G} \ S_c.next$$
$$\Rightarrow S_a \, {}^{I}\!\underset{\sim}{\sqsubset}{}^{O}_{*} \, S_c$$

Example As an example, we can consider the operation `diff` of the bag example. The input and the output of `diff` are redefined as bounded sequences or the special overflow constant.

We can define `I` and `O` as follows:

$$\begin{aligned} I(i,j) &\equiv j \neq \mathit{Oflow} \Rightarrow i = \mathit{SeqToBag}(j) \\ O(i,j,o,p) &\equiv p \neq \mathit{Oflow} \Rightarrow o = \mathit{SeqToBag}(p) \end{aligned}$$

This correspondence is correct if overflow is observable. The output of the concrete `diff` operation is thus supposed to return overflow if the input or the internal state has the *Oflow* value.

5.2 Retrenchments as sufficient conditions for conditional refinements

This paragraph presents another instance of the general definition of retrenchment which defines a notion of conditional refinement. In order to be compatible with the trace point of view, the condition does not depend on the internal concrete or abstract states. This notion could be combined with input/output conversion, but we present it separately.

The following definition introduces trace refinement conditioned by an always true property. Two equivalent definitions are given, the second one being closer to retrenchment: it uses a `concedes` predicate.

Definition 8 (Conditional trace refinement)

$$\begin{aligned} S_a \sqsubseteq S_c/P &\equiv \forall\, i,o : (\Box P(i,o)) \Rightarrow \overline{S_c}(i,o) \Rightarrow \overline{S_a}(i,o)) \\ &\equiv \forall\, i,o : \overline{S_c}(i,o) \Rightarrow \overline{S_a}(i,o) \vee \Diamond\neg P(i,o) \end{aligned}$$

The following theorem introduces retrenchment as a sufficient condition for conditional refinement.

Theorem 4 (Sufficient condition for conditional trace refinement)

$$(\forall v : S_c.init(v) \Rightarrow \exists u : G(u,v) \wedge S_a.init(u)) \wedge S_a.op \ {}_{\neg P}\underset{\sim}{\sqsubset}{}_{G} \ S_c.op \Rightarrow S_a \sqsubseteq S_c/P$$

Example The implementation of an infinite bag by a bounded sequence does not define a refinement, but such an implementation is acceptable if one can establish that the sequence does not overflow when integrated in the considered system. This property can be specified by a conditional trace refinement property. For this purpose, overflow must be observed. We suppose that adding an element to a bounded sequence returns a status (done, ovf).

$$\forall\, i,o : (\Box o \neq \mathrm{ovf}) \Rightarrow (\overline{S_c}(i,o) \Rightarrow \overline{S_a}(i,o))$$

5.3 Recovering synchronization

We consider an infinite buffer as abstract model and a lossy infinite buffer as concrete model. The abstract input trace contains `put` and `get` operation calls. The concrete input trace can also contain `Loss` operations. Given a concrete execution trace, the following trace level specification claims the existence of an abstract execution trace obtained by replacing `Loss` calls by `get` calls. As losses take place where `get` operations could be performed, losses occurs on the output side of the channel:

$$\forall\, i, j, p : \overline{S_c}(j, p) \wedge \Box(i = j \vee (j = \mathrm{Loss} \wedge i = \mathrm{get})) \Rightarrow \exists\, o : \overline{S_a}(i, o) \wedge \Box(j \neq \mathrm{Loss} \Rightarrow o = p)$$

Considering retrenchment, this property can be expressed using the identity as gluing invariant: the hidden states defining the contents of the abstract and concrete channels are identical. Input and output conversions are used to refine a `Get` into a `Loss`.

Losing synchronization Consider again the infinite buffer example and its bounded implementation by a finite sequence. `get` is supposed to return `ovf` once an overflow has been detected. Thus, synchronization between the concrete and abstract traces is lost. This can be expressed by the following trace refinement property with output conversion, using the *weak until* temporal operator.

$$\forall\, j, p : \overline{S_c}(j, p) \Rightarrow \exists\, o : (p = o \ \mathbf{W}\ p = \mathrm{ovf}) \wedge \overline{S_a}(j, o))$$

Recovering synchronization. Assume the existence of a `reset` operation which empties the bag and thus allows a recovery of synchronization. The new behavior can be specified as follows:

$$\forall\, j, p : \overline{S_c}(j, p) \Rightarrow \exists\, o : \Box(j = \mathrm{reset} \Rightarrow p = o \ \mathbf{W}\ p = \mathrm{ovf}) \wedge \overline{S_a}(j, o))$$

The conditional *weak until* expression can be eliminated with the introduction of an auxiliary state variable: $p = o$ must be true since $j = \mathrm{reset}$ has been true and until $p = \mathrm{ovf}$ becomes true. The auxiliary variable m is defined to be true during this time interval.

$$\begin{array}{ll} \forall\, j, p, m : \overline{S_c}(j, p) \wedge m \wedge \Box(m' = & \mathbf{IF}\ j = \mathrm{reset}\ \mathbf{THEN}\ \mathit{true} \\ & \mathbf{ELSE\ IF}\ p = \mathrm{ovf}\ \mathbf{THEN}\ \mathit{false} \\ & \mathbf{ELSE}\ m) \\ \quad \Rightarrow \exists\, o : \Box(m \Rightarrow o = p) \wedge \overline{S_a}(j, o)) & \end{array}$$

Once again, refinement with output conversion is a sufficient condition for this temporal specification. However, the variable m must be added to the concrete model.

Remark The transformation illustrated here becomes a proof method for the conditional weak until operator that is applicable to B specifications. It amounts to adding a new state variable.

6 Conclusion

Refinement gives a strategy for specifying and developing a system which conforms to its specification: an abstract machine expressing the desired behavior is defined. Then, by stepwise refinements, a certified implementation is built. Retrenchment can be seen as a new approach to implement a correct system which extends refinement: the concrete system is created by specifying permitted differences from the abstract system.

In this paper, this differential technique is used for combining two styles of specification: abstract machine and temporal specifications. We have explored some of the potential of this methodology. Starting from a perfect abstract system and a temporal formula expressing differences between perfect and effective input/output traces, a concrete system is proposed together with a gluing invariant relating the abstract and concrete state spaces. Then, retrenchment provides a proof obligation schema to validate the concrete model. We can see how this work can be used as a way to specify safety properties of a retrenched implementation. A natural outgrowth of this work would be to extend it in order to take into account liveness properties.

References

1. J. Abrial. *The B-Book: Assigning programs to meanings.* Cambridge University Press, 1996.
2. A. Arnold. *Finite transition systems.* Prentice-Hall. Prentice-Hall, 1994.
3. R.-J. J. Back, A. Akademi, and J. V. Wright. *Refinement Calculus: A Systematic Introduction.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1998.
4. R. Banach and M. Poppleton. Retrenchment. In *FM '99: Proceedings of the Wold Congress on Formal Methods in the Development of Computing Systems-Volume II*, pages 1864–1865, London, UK, 1999. Springer-Verlag.
5. R. Banach and M. Poppleton. Retrenchment, refinement and simulation. In J. Bowen, S. King, S. Dunne, and A. Galloway, editors, *Proc. ZB2000*, volume 1878 of *Lecture Notes in Computer Science*, York, September 2000. Springer.
6. W.-P. de Roever and K. Engelhardt. *Data Refinement Model-Oriented Proof methods and their Comparison.* Cambridge University Press, 1998.
7. L. Lamport. *Specifying Systems:The TLA+ Language and Tools for Hardware and Software Engineers.* Addison-Wesley, 2002.
8. N. Lynch and M. Tuttle. An introduction to I/O automata. *CWI-Quarterly*, 3(2):219–246, sept 1989.