A PVS specification of an invoicing system*

Heinrich Rust**

Lehrstuhl für Software Systemtechnik, BTU Cottbus

Abstract. We present a PVS specification of an invoicing system. We use PVS's features to structure the specification, to define types and operations on them. We put special emphasis on PVS's capabilities to add information to a specification meant to be redundant, and to check that this information for consistency. We heavily use the possibility to define the type of some argument to a function depending on values of other arguments to the function, and to let the system automatically generate lemmas to be proved for checking consistency of function declaration and usage. PVS is found to be helpful because of the automatic support for doing proofs.

1 Introduction

This paper reports work done on a case study proposed by a group at IRIN in Nantes, France. A simple invoicing system was to be specified. The case study description presuppose some knowledge about the application field. Questions like the following have to be answered: What does it mean for the stock when an order is invoiced or cancelled? Which kind of orders may be cancelled? Regarding questions like these, a lot of decisions have to be taken when a precise specification is fixed. Some of these questions will be answered wrongly by the specifier, and for some of these questions, the customer will not have an answer ready. Feedback from the customer about an unambiguous specification is the most efficient way to find out about errors, misunderstandings and vagueness. In this work, we develop a specification for one understanding of the informal description. This would be the base for discussions with the customer and this discussion would, typically, not only make the specification better but also show errors in the informal description.

The case study description discusses two cases which are to be considered when the system is specified. We understand the two cases in the way that a) it should be possible to analyse a configuration of an invoicing system consisting of a fixed stock and a fixed set of orders, and b) to allow operations on such a configuration like addition of new orders, cancellation of orders, and updates of stock content for product types. These two cases are not mutually exclusive. We will specify our understanding of the static structure (which corresponds to case 1) and of the dynamics of the system (corresponding to case 2). The static structure is modeled via the definition of data types, and the

* Presented at the invoice-workshop 1998 in Nantes.

** BTU, Postfach 101344, D-03013 Cottbus, Germany; Tel. +49(355)69-3803, Fax.:-3810; rust@informatik.tu-cottbus.de

dynamics via the specification of operations with their preconditions and effects: invoicing an order in the system, introduction of a new order into the system, cancellation of an order, and updating the stock content for a product type.

We use PVS as a specification notation and proof support system [ORSvH95]. PVS specifications are grouped in "theories". These are namespaces which contain type definitions, value definitions (which include values of higher logical types, and which can use an axiomatic or an applicative style), and theorems which are expected to be provable from the axioms. PVS has a very powerful type system: From every definable predicate, a type can be constructed. This leads to a drawback: Type correctness is not decidable for this specification language. This problem is solved in PVS in the following way: For a given specification, the PVS system generates a set of lemmata which must be proved to ensure type correctness of the specification. These automatically generated lemmata are called 'type correctness conditions', or TCCs. We will sometimes refer to them as to proof obligations. This automatic generation of proof obligations is the main feature to be demonstrated in this approach. Together with the powerful proof support system, it is the main advantage in comparison to, for example, Z [Spi92], which might allow otherwise similar specifications.

Like using Z, we can stay quite abstract when using PVS as a specification language. We do not have to specify an implementation. And also like using Z, we can build on sets to model the data types and operations. PVS is more programming language like than Z in that is distinguishes types and namespaces syntactically. Like PVS specifications, those done in a programming languages also have the benefit of allowing automatic and semi-automatic checks for those properties which can be expressed as type correctness properties of programs. PVS' advantage is in this respect that the type system is far more flexible: It includes a parser and a proof support system with the help of which type correctness can be checked like in a programming language, with the extension that we may also define types for which inclusion of expressions is not decidable automatically. For these cases, the PVS system automatically constructs the TCCs mentioned.

We distinguish three categories of questions which we had to answer to develop a specification for the invoice system:

- Some questions regard the mathematical model of the specification domain. These questions are indirectly evoked by our choice of the specification formalism. In PVS, we are forced to model the application area concepts as types and functions. This is the only set of questions which is application domain specific.
- Other questions regard the readability of the specification. This concerns global structure, layout and the like. These questions are not application domain specific.
- A last category of questions deal with usage of PVS pecularities. These questions are also not application domain specific.

Mathematical properties The first set of questions regards the mathematical model of the specification domain. This are the following:

- What are the object types in the world of discourse?
- What are the operations on objects of these types?
- How do the operations on objects work? How are they mathematically described?

These questions are not specific to the usage of PVS for the specification. They appear in every method in which an application domain is modeled as consisting of a number of basic object types and a number of operations on them. It is interesting how well the mathematical concepts are expressed in the PVS notation, i.e. if the encoding of the intuitive concepts in the specification notation at hand is difficult or not.

Readability The following questions regard readability of the specification text:

- What should be the global structure of the specification?
- What conventions should be used for identifiers?
- How do we deal with layout, indentation and the like?

These question also have to be answered for any specification notation. They are sometimes not emphasized enough: Readability, and thus structure, are the most important preconditions for a specification to be checkable.

PVS specifics The last set of questions regard PVS specifics:

- How should we structure the specification?
- How should we model the basic types?
- How should we model effects and preconditions for the application of operations?
- Should we use an axiomatic or an applicative style for the definition of functions? We can avoid consistency problems by using an applicative style.
- Should we use specialized but less easy to understand concepts like dependent types or not? Since we want to show special possibilities of PVS, we will not avoid the more complicated concepts.
- What strategies should we follow for inclusion of redundancy into the specification?

Our decisions concerning all these questions will become clear during our presentation of the specification.

2 A complete PVS specification of the invoicing system

In this section, we present our PVS-specification of the invoicing system. We will go through this specification sequentially and describe the PVS concepts used when we deal with the concepts modeled with their help.

2.1 The structure of the specification

PVS specifications are structured into 'theories'. Theories are packages of declarations of types, of values belonging to these types, and of axioms and theorems expressing properties of these values. Theories are typically used to collect type and object declarations which belong together and the axioms and theorems belonging to them. We use identifiers starting with `Th` and continuing with a capitalized word.

We decided to distribute the declarations into three theories:

– `ThOrderSet` contains type declarations and definitions for orders and sets of orders.
  Most properties of orders are suggested nicely from the case study description. Exceptions are order IDs and consistent order sets. The latter is the prime concept we introduce for modeling the invoicing system. Such a set encompasses a set of orders, where each order is in one of several states. The use of order sets is one of the major design decisions we took.
– `ThStock` contains declarations for stocks. This builds on some elements from `ThOrderSet`. Stocks are modeled as functions from item types to quantities. Here, we collect functions modeling operations and predicates concerning an order in relation to a stock.
– `ThConfig` contains declarations to be used for configurations. This builds on both `ThOrderSet` and `ThStock`. A configuration models a stock and an associated set of orders. Operations like adding or cancelling an order are modeled via functions on configurations.

2.2 Types for the definition of consistent order sets

PVS allows to define new types. In this respect, it is similar to typed programming languages.

There are several possibilities to define types in PVS. For the definition of consistent order sets, we use some of these constructs, cf. Fig. 1. We will explain the concepts when we discuss the type declarations in which they are used.

For type identifiers, we use the following notation: They start with a capital `T`, then follows a capitalized name.

– `TOrderId` and `TItemType` are uninterpreted nonempty types. 'Uninterpreted' means that we do not define a structure for this type. It is just a set of elements. The plus-sign behind the keyword `TYPE` denotes that this set is nonempty.
  The decision to use elements of a special type `TOrderId` to identify orders is a critical one in our specification. We will call these elements "order IDs". The components of an order as described in the case study description, can not be used to construct a key for orders. It will make the specification of the operations to be defined conceptually simpler to have a key for orders. Therefore we decided to extend the application domain by one more type, against the advice in the task description of the case study. The main property of order IDs in relation to sets of orders, i.e. that they occur at most once, is expressed in the type definition `TConsOrderSet`. We will describe the PVS-specification of this property when we deal with the definition of that type.
  `TItemType` is a type for the different types of objects which can be in a stock and which can be referenced in an order. As with `TOrderId`, we do not define an internal structure or further properties for elements of this type.
  PVS allows to model these types very abstractly. We do not have to give structure for these types.
– `TQuantity`: Elements of type `TQuantity` are used in orders and in stocks to define the quantities ordered or in stock for a given item type. We model these quantities as non-negative real numbers.

```
ThOrderSet: THEORY
BEGIN
    % An uninterpreted nonempty type for order ids.
    TOrderId: TYPE+

    % An uninterpreted nonempty type for item types.
    TItemType: TYPE+

    % A type for quantities.  Quantities are the nonnegative reals.
    TQuantity: TYPE = { r: real | r >= 0 }

    % An enumerated type for the state of an order.  It can be
    %  pending or invoiced.
    TOrderState: TYPE = { osPending, osInvoiced }

    % A record type for orders.  An order consists of
    %  an id, the item type, a quantity and a state.
    TOrder: TYPE =
        [#  oid: TOrderId, itemType: TItemType,
            quantity: TQuantity, orderState: TOrderState
        #]

    % A type for the pending orders.
    TPendingOrder: TYPE =
        { order: TOrder | orderState(order) = osPending }

    % A type for consistent sets of orders.  Each order id
    %   is used in such a set for at most one order.
    TConsOrderSet: TYPE =
        {
            orderSet: setof[TOrder]
        | FORALL(order1,order2: TOrder):
            ( member(order1,orderSet) AND member(order2,orderSet) )
                IMPLIES
            ( (order1 = order2) OR NOT (oid(order1) = oid(order2)) )
        }
```

Fig. 1. Theory `ThOrderSet`, part 1/2: Type definitions

In our specification, this type is defined to be a subtype of an existing type, the real numbers. There are several predefined types in PVS. The real numbers, written `real`, the natural numbers, written `nat`, and the booleans, written `bool`, are some one of them.

We want `TQuantity` to allow all non-negative reals. We can specify this type with a set-like notation which resembles the notation with a base set and a defining predicate known from mathematics.

On the basis of the given case description, it can not be decided if it is more appropriate to model quantities as naturals, for example. We decided to use a fairly general approach.

The real type and the natural or integer types have a semantics in PVS like in ordinary mathematics, that is, there are no restrictions resulting from finiteness of a representation, like the value ranges or the accuracy. This is in contrast to typical programming languages.

– `TOrderState` is used to define different possible states an order can be in. We decided, based on the description in the case study, to model this with an enumerated type with two elements, one for a pending order, and one for an order which has been invoiced. These two elements are called `osPending` and `osInvoiced`. We decided not to use a special state for canceled orders. We will represent the effect of canceling an order via removal of the order from a set encompassing all pending and invoiced orders. This decision is made explicit in this type definition.
– The types defined so far are used in the definition of the order type `TOrder`. We specify this as a record type. In PVS, this is not meant to imply an implementation for the type, but describes some mathematical properties for elements. For example, we know that for an order, we have four functions yielding its associated ID, the item type, the quantity and the state, and that two orders for which these four pieces of information are equal, are themselves equal.
  In PVS, a functional notation is used for access to components of a record.
– `TPendingOrder`: This is a subtype basing on the type `TOrder`. In a set-like syntax, it is defined that `TPendingOrder` is a type encompassing those `TOrder`-individuals whose order state is pending.
– `TConsOrderSet`: This is another subtype, a bit more complicated than the one defined before. The base type `setof[TOrder]` from which this subtype is derived denotes the type encompassing all sets of orders.
  Set types are not basic in PVS. They are implemented as boolean function types yielding 'true' for the elements of the represented set and 'false' for its complement. Set-notation is just an alternative to notations based on boolean functions. A function type from a type `A` to a type `B` is written `[A -> B]`. `setof[A]` is just an alternative for `[A -> bool]`.
  Let us call an order set consistent iff each order ID occurs in at most one order of the set. `TConsOrderSet` is defined to encompass each individual of type `setof[TOrder]`, i.e. each set of orders, which is consistent. This is defined behind the bar of the subtype declaration of `TConsOrderSet`. Boolean operators (e.g. `AND`, `IMPLIES`, `OR`, `NOT`), set functions (`member(elem,set)`) and equality are used to define the consistent order sets.
  Our reason to introduce order IDs was to be able to define consistent order sets.

2.3 Functions involving consistent order sets

The types described in the preceding section are used to define functions involving consistent order sets. This is shown in Fig. 2. PVS is a higher level specification notation. Constants and functions do not belong to different syntactical categories, and a function may get functions as arguments and may return functions as results.

For the definition of functions, there are several alternatives in PVS. We might use an axiomatic style in which we introduce the function identifier as belonging to a given type, and in which we describe in some axioms which properties the function identified by the identifier has. The problem is that inconsistencies in the axioms may be unnoticed. Because of this, there is the alternative to define functions in an applicative style: We specify how we could compute the result of the function application to its arguments. This also defines a function. The applicative definition of a function might

```
    % Determine the set of pending orders in an order set.
    pendingOrders(orderSet: TConsOrderSet): TConsOrderSet =
        { order: TPendingOrder | member(order, orderSet) }

    % Determine set of order ids used in an order set.
    usedOrderIds(orderSet: TConsOrderSet): setof[TOrderId] =
        { orderId : TOrderId
        |
            EXISTS (order: TOrder):
                member(order, orderSet) AND (oid(order) = orderId)
        }

    % Check if the orderId is used in the given order set.
    orderIdIsUsedInOrderSet?(orderId: TOrderId)(orderSet: TConsOrderSet)
       : bool = member(orderId, usedOrderIds(orderSet))

    % Fetch the order with the given id from the order set.
    getOrderForId(
        orderId: TOrderId,
        orderSet: (orderIdIsUsedInOrderSet?(orderId))
    ) : TOrder =
        % Select the element from the orderSet for which
        %  the id is orderId, if one exists.
        epsilon( {order : (orderSet) | oid(order) = orderId } )

    % Change the state of the order referenced by the given id,
    %  if possible.  If not, return order set unchanged.
    changeOrderState
        (orderId: TOrderId, orderState: TOrderState)
        (orderSet: (orderIdIsUsedInOrderSet?(orderId)))
    : TConsOrderSet =
        LET
            % Get the order for the argument id.
            idOrder = getOrderForId(orderId, orderSet)
        IN
            % Add order with changed state to order set from
            %   which old order has been removed.
            add(
                idOrder WITH [(orderState) := orderState],
                remove(idOrder,orderSet)
            )
END ThOrderSet
```

Fig. 2. Theory `ThOrderSet`, part 2/2: Operations

suggest an implementation. It is important no note that this implementation aspect is not encompassed by the semantics of a PVS function specification. Two identical functions with different applicative definitions are considered to be identical in PVS.

We explain the declarations of our functions involving consistent order sets:

– `pendingOrders`: This is a typical applicative function definition. The syntax is similar to the one used in a lot programming languages. The function identifier and a formal parameter list are followed by a return type. After the equal sign, an expression is given which, evaluated for the given arguments, yields the value of the function application.

This function is defined to yield the subset of pending orders of a given consistent order set. We use the set notation to define this function.

- `usedOrderIdents`: The set of used order IDs for a given set of orders is defined. Here, we use the existence quantor `EXISTS` for the definition of the function. PVS also has the `FORALL` quantor.
- `orderIdentIsUsedInOrderSet?`: The question mark is a valid letter in identifiers. The PVS convention is to use it as the last letter in identifiers for boolean functions.
  This function is designed to be usable in two ways. We can, for a given order ID and a given order set, determine if the order id occurs in an order of the order set, and we can express, for a given order ID, the set of all order sets in which a given order ID occurs. This double use is made possible by two PVS features: One is the already mentioned implementation of sets as boolean functions, the other is the possibility to define functions resulting in functions.
  We specified `orderIdentIsUsedInOrderSet?` as a higher level function. It has the type `[TOrderId -> [TConsOrderSet -> bool]]`, which is the same as `[TOrderId -> setof[TConsOrderSet]]`.
  The two possible uses are differentiated by applying the function to just an order ID, which results in a set, or to apply it first to an order ID and to apply the resulting function to an order set, which results in a boolean. Thus, `orderIdentIsUsedInOrderSet?(orderIdent)` represents the set of all sets of consistent orders in which `orderIdent` occurs, and `orderIdIsUsedInOrderSet?(orderIdent)(orderSet)` is true iff `orderIdent` occurs in an order of `orderSet`.
  The function is defined via the `member`-function which comes from a library of predefined functions. We use the previously defined function `usedOrderIdents` for a concise definition of the function.
- The function `getOrderForIdent` results in the order which is named by a given ID in an order set. Here, we have to solve the problem how to deal with exceptional conditions: What should be returned if the order ID does not occur in the order set? There are several solutions to this problem:
  - We might yield a special error value. This would be possible in PVS.
  - Implement some kind of exceptions which interrupt the standard program flow. This is not possible in PVS. PVS functions are mathematical functions, not programming language functions.
  - Define the function as partial and let the values for illegal arguments be undefined. This would be a possibility in Z, but in PVS, functions are always total.
  - Check that the function is never applied when no result can be given. This is the alternative we will choose, and which is very nicely supported by PVS.

  Several features of PVS are used in our approach to solving the problem:
  - We can define functions with two arguments, yielding a `TOrder`. There is a difference between type `[A,B -> C]` and `[A -> [B -> C]]`. For `getOrderForIdent`, we use the former type.
  - A set expression in parentheses denotes a type. We use the set expression `orderIdentIsUsedInOrderSet?(orderIdent)` to define a type consisting of just those consistent order sets in which `orderIdent` occurs.

- The type of the second argument (`orderSet`) depends on the value of the first argument (`orderIdent`). These so called 'dependent types' are a special feature of PVS.
- In the definition of the function, Hilbert's epsilon-operator is used to access a set whose representation is not fixed. Its meaning is: If there is an element in the set, return it. Otherwise, return any element from the base set.

The PVS system will try to ensure that any usage of this function is type correct. If it cannot do this automatically, it will generate a lemma to be proved by the specifier which ensures that the second argument in a function usage belongs to the specified type.

– The function `changeOrderState` is used to model an operation on an order set which changes the state of an order in the set. The order is specified via an order ID.
We decided to specify this kind of operations on an object, like an order set in our case, as higher level functions in which one of the arguments is of the object type manipulated, and in which the resulting object contains the changed object.
Higher level functions are used to make explicit which particular roles the different arguments play. `orderIdent` and `orderState` are ordinary arguments of the operation. The operation operates on an order set. This is given as an argument to the function resulting from `changeOrderState(orderIdent,orderState)`.
How will we deal with illegal arguments in operations? We could choose to just ignore these operations, i.e. return the manipulated object unchanged, or we could choose some of the strategies enumerated previously. We will again use the approach described above: We will use dependent types to check that operations are only applied when they are can be performed. For this, we do not allow every consistent order set as an argument to the function. Only those for which the operation can be performed are allowed. We use a dependent type to ensure that the argument order set always contains an order for the order ID given as first argument.
The `LET`-expression allows to use the identifier `idOrder` as an abbreviation for the value denoted by `getOrderForIdent(orderIdent, orderSet)`.
The `add`-expression constructs the changed order set:
  - The `remove`-expression yields an order set from which the order to change has been removed.
  - The `WITH`-expression takes the order denoted by `idOrder` and constructs a variant. In this variant, the value of the `orderState`-field has been changed to the order state given as argument.
    The value of a `WITH`-expression is equal to the first argument of the expression (here: `idOrder`), where some components might be changed. These changes are given in a bracketed assignment.
  - The `add`-expression constructs a set which contains the element given as first arguments and all arguments of the set given as second argument.

2.4 Stocks and their operations

The second theory collects declarations concerning stocks. It is displayed in Fig. 3. It uses declarations from the theory `ThOrderSet` via importing this theory.

```
ThStock: THEORY
BEGIN
    IMPORTING ThOrderSet

    % A stock maps an item type to a quantity.
    TStock: TYPE = [TItemType -> TQuantity]

    % Check, for a given pending order, if it can be invoiced
    %   for a given stock.
    canInvoice?
        (pendingOrder: TPendingOrder)
        (stock: TStock)
    : bool =
        (quantity(pendingOrder) <= stock(itemType(pendingOrder)))

    % Perform an invoice for a given order on a given stock.
    doInvoice
        (pendingOrder: TPendingOrder)
        (stock: (canInvoice?(pendingOrder)))
    : TStock =
        % Diminish stock content for the item type by the given quantity.
        stock WITH
            [
                (itemType(pendingOrder))
                    :=
                (stock(itemType(pendingOrder)) - quantity(pendingOrder))
            ]
END ThStock
```

Fig. 3. Theory for stocks and their operations

– We define the type `TStock` as a function type. Every item type is assigned a quantity by functions of this type.

– The predicate `canInvoice?` checks if a given pending order can be invoiced in a given stock. This is the case if the quantity in stock is at least as large as the quantity ordered.

– `doInvoice` is an operation on stocks. We use a higher level type again. The argument to the operation is a pending order. We will not allow to use this operation when the invoice can not be performed. This can be checked automatically by restriction of the stocks to which the operation may be applied to those for which this is possible. The type `(canInvoice?(pendingOrder))` based on the function `canInvoice?` characterizes this. The `WITH`-expression constructs the new stock on the base of the old one by yielding for the item type in the given order a quantity diminished by the quantity in the argument order.

  The basic assumptions regarding the mathematics of invoicing an order go into our specification in the definition of this function.

```
ThConfig: THEORY
BEGIN
    IMPORTING ThOrderSet, ThStock

    % A type for a configuration of the ordering system
    %    consisting of a stock and a consistent set of orders.
    TConfig: TYPE = [# stock: TStock, orders: TConsOrderSet #]

    % Is a given orderId used in a given configuration?
    orderIdIsUsedInConfig?(orderId: TOrderId)(config: TConfig)
    : bool = orderIdIsUsedInOrderSet?(orderId)(orders(config))

    % Does the orderId belong to a pending order in the given
    %  configuration?
    orderIdIsPending?
        (orderId: TOrderId)
        (config: (orderIdIsUsedInConfig?(orderId)))
    : bool =
        (osPending = orderState(getOrderForId(orderId, orders(config))))

    % Is the order belonging to orderId invoiceable in the given
    %  configuration?
    orderIdIsInvoiceable?
        (orderId: TOrderId)
        (config: (orderIdIsPending?(orderId)))
    : bool =
        canInvoice?
            (getOrderForId(orderId, orders(config)))
            (stock(config))
```

Fig. 4. Theory for configurations, part 1/3

2.5 Types and operations for configurations

The last theory `ThConfig` (Fig. 4 and Fig. 5) combines declarations from the other ones and defines on top of them a type and some functions for configurations consisting of a stock and an order set.

– `TConfig` is a record type defining configurations. Such a configuration contains a stock and a set of consistent orders.

We define some boolean functions to describe sets of configurations which fulfill special conditions (Fig. 4):

– `orderIdentIsUsedInConfig?`: This predicate characterizes configurations in which an order with the given ID occurs.
– `orderIdentIsPending?`: This predicate characterizes configurations in which an order for the given order ID occurs and is pending.
– `orderIdentIsInvoiceable?`: This predicate characterizes configurations in which an order for the given order ID occurs, this order is pending, and this order is invoiceable for the given stock.

```
% Invoice the order referenced by the given orderId.
%   Allow only configurations where the operation is legal.
invoiceOrder
    (orderId: TOrderId)
    (config: (orderIdIsInvoiceable?(orderId)))
: TConfig =
    % Construct the configuration after the invoice.
    (#
        % Construct the stock after the invoice.
        stock := doInvoice(getOrderForId(orderId, orders(config)))
                          (stock(config)),

        % Construct the order set after the invoice.
        %  Change state of idOrder to invoiced.
        orders := changeOrderState(orderId, osInvoiced)
                                  (orders(config))
    #)

% Insert an order into a configuration.
%  We use a dependent type to specify that the configuration may not
%  use the id of the given order.
insertOrder
    (order: TPendingOrder)
    (config: (difference(fullset,
                         orderIdIsUsedInConfig?(oid(order)))))
: TConfig =
    % Insert given order to order set.
    config WITH
        [ orders := add(order, orders(config)) ]

% Set the quantity of the given itemType to the given value.
setQuantity
    (itemType: TItemType, quantity: TQuantity)
    (config: TConfig)
: TConfig =
    % Change the quantity of the given item type
    %   in the stock of the configuration.
    config WITH
        [ stock := (stock(config) WITH [itemType := quantity]) ]
```

Fig. 5. Theory for configurations, part 2/3

Finally, we define some operations for the execution of operations on configurations, one for invoicing orders, inserting orders, one for canceling orders, and one for changing the stock content for a given item type (Fig. 5 and Fig. 6). Here, the correspondence of the case study descriptions and our specification is especially strong. From the case study description, we infer a set of operations which should be possible in the invoicing system. These are modeled as operating on configurations, and they are specified as higher level functions.

– `invoiceOrder`: This is an operation on a configuration which yields a configuration in which one pending order has been invoiced. The configuration given to this operation is specified to belong to the set of configurations for which the invoicing operation is possible with the order associated with the argument ID.

```
    % Cancel an order.
    %  We use a dependent type to specify that the configuration must
    %   include an order with the given id.
    cancelOrder
        (orderId: TOrderId)
        (config: (orderIdIsUsedInConfig?(orderId)))
    : TConfig =
        % The id of the order to be cancelled occurs in
        %  the order set of the configuration.  Now we have to
        %  check if a pending or an already invoiced order
        %  is canceled.
        LET
            % Determine the order for the given id.
            idOrder = getOrderForId(orderId, orders(config)),
            % Construct order set without the order referenced.
            smallerOrderset = remove(idOrder, orders(config))
        IN
            % Check if the order is pending or invoiced.
            CASES orderState(idOrder) OF

                osPending:
                    % Just remove the order from the order set.
                    %  Stock must not be updated in this case.
                    config WITH
                        [ orders := smallerOrderset ]

                osInvoiced:
                    % Add quantity of canceled order to
                    %   stock and remove order from order set.
                    (#
                        stock := stock(config) WITH
                            [
                                itemType(idOrder) :=
                                      stock(config)(itemType(idOrder))
                                    + quantity(idOrder)
                            ],
                        orders := smallerOrderset
                    #)
            ENDCASES

END ThConfig
```

Fig. 6. Theory for configurations, part 3/3

The expression starting with `(#` and ending with `#)` is a record expression constructing a configuration from a stock and an order set based on the previous stock and order set. The new stock is constructed by applying the `doInvoice`-function from the theory `ThStock` to the stock, and the new order set is constructed by changing the state of the order referenced in the argument from `osPending` to `osInvoiced`.

– `insertOrder`: We can insert a pending order into an order set in which the ID of the pending order does not yet occur. This precondition is encoded via the types of the arguments to the function: The order must be of type `TPendingOrder`, and the configuration is a type dependent on the ID of this order. This type is,

via the parenthesis-notation, constructed from a set expression. The set expression characterizes the configurations where the ID of the given order does not occur in the order set components.

– `setQuantity`: This last operation on configurations just sets the quantity of the given item type to the given number. A nested `WITH`-expression is used for the specification of the result of this operation, an outer `WITH` for the change of the stock in the configuration, and an inner `WITH` for the change of the quantity entry on this stock for the item type whose quantity is set in the operation.
– `cancelOrder`: This operation is used to remove an order characterized by a given order ID from the order set of a configuration. Here again, we use the dependent type mechanism for the arguments of the function to specify the allowed arguments: We allow only configurations in which the order ID given as first argument does occur.

  The function is specified via a `LET`-expression in which we assign a name to the order referenced by the argument ID and to the order set which results from removed this order from the order set of the configuration. After the keyword `IN`, we give an expression for the value of the function.

  We use a case distinction based on the state of the order being canceled. This expression makes our understanding of the effect of canceling an order explicit: If the order is pending, the resulting configuration is the old one in which the order set is only replaced by the reduced order set. If the order being canceled already has been invoiced, we update the stock by adding the quantity of the order to the stock content for the referenced item type.

3 Verifications

In our context, the term "verification of formal specifications" is used for checks that the informal concepts a specifier has in mind are consistent with the formalized specifications written down. A powerful way to check this is to put redundant information into the written specification. These pieces of information meant to be redundant should not be based on each other but on the specifier's intuitive concepts. Verifications in this sense can not be totally formal because of the informal nature of mental concepts.

The direct check that the written specification conforms to the informal ideas can be made easier by using a notation and a style of layout which makes it easy to understand a written specification. This is the basic way to verify a specification. This strategy is not specific to PVS.

Entry of redundant information and consistency checks are supported by PVS in several ways:

– The PVS parser finds purely syntactical errors which not only indicate oversights, but also sometimes conceptual inconsistencies or misunderstandings.
– Argument types of functions can be defined exactly for the allowed expression types via the dependent type mechanism. The generation of TCCs focuses the attention on problematic items.
– Lemmata which the specifier expects to be true can be added to the system.

– The proofs of both the automatically generated lemmata and the ones added by hand are supported by a system which helps the specifier in doing the easier steps automatically, to do bookkeeping about proofs yet to be done, and to rerun proofs after changes in the specification.

In our approach, we tried to put as much information as possible into the type declarations of the functions. In this way, consistency checking proof obligations are generated automatically, and it is not necessary to check the basic properties with explicitly given lemmas.

What results did we get? Type-checking our specification resulted in about a dozen TCCs all of which could be proved easily with PVS's proof support system. This consistency check depends on PVS to find all necessary TCCs.

If compared, for example, with Z: What are the advantages to use PVS for a specification? Our approach illustrated the following:

– Several structuring mechanisms to bundle information and to be able to refer to it via an identifier are supported (theories and types), and a clear name space mechanism is provided.
– The powerful type system together with the automatic generation of TCCs, the possibility to define functions applicatively and the proof support system helped to increase trust in the consistency of the specification.

4 Summary

Formulating the specification with mathematical precision forced us to decide exactly how the stock and the order set interact, which operations we need and which preconditions have to be fulfilled for an operation. Internal consistency was checked with the possibility to use dependent types and to let PVS generate proof obligations automatically.

Acknowledgements

I thank Claus Lewerentz for discussions with him regarding both form and content of this paper.

References

[ORSvH95] Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.

[Spi92] J. M. Spivey. *The Z Notation*. Prentice Hall, Hemel Hempstead, 2 edition, 1992.