Mixed Formal Specifications with PVS

Michel Allemand and Jean-Claude Royer
IRIN, Université de Nantes
2 rue de la Houssinière, B.P. 92208, F-44322 Nantes cedex 3, France
(Michel.Allemand,Jean-Claude.Royer)@irin.univ-nantes.fr

Abstract

Formal specification of mixed systems is one of the main issues in software engineering. However several difficulties remain. Amongst them is the ability to produce a coherent mixed specification and to provide tools for verifying it. The Graphic Abstract data Type (GAT) approach is proposed to cope with this issue. GAT is a mixed formalism based on Symbolic Transition Systems (STSs) and algebraic specifications of partial abstract data types. This paper presents how to translate a GAT into PVS. The principle is to define a total data type with generators and predicates. Then a partial data type is obtained using the previous one and the definedness predicate as a PVS subtype predicate. This allows us to get the benefit of the PVS automatic generation of type-correctness conditions. Several ways to translate operation axioms are discussed. Lastly our paper shows some proofs of temporal properties.

1. Introduction

Formal specifications are now widely accepted but there are still some difficulties related to their use. One of them is to specify systems with static, dynamic and functional aspects, we call them mixed systems. These are complex and realistic systems where the use of several formalisms is required, for instance with architectural description languages [9]. The two main issues with mixed systems are first to ensure the consistency between the different aspects and second to provide specification and proof guidelines. We focus in this paper on the use of theorem provers to support mixed systems.

Several approaches for mixed systems have been studied in the area of algebraic specifications. A full comparison with existing works would be too long, related works are [5, 2, 6, 4]. We have chosen an integrated approach [13] with a single semantic framework based on partial algebras [17]. To specify mixed systems, we proposed the Graphical Abstract Data Type concept [14]: a symbolic transition system coupled with an algebraic specification. For each component we consider two views: the dynamic view and the functional view. The dynamic view describes some static features and the dynamic part of the component. It uses a notion of finite and Symbolic Transition System (STS). For related work see [8, 16]. The functional description is an algebraic specification of a partial abstract data type. The STS part is formally described as operations and axioms in this partial data type. Proofs in our context need only one formalism and one environment.

A related approach is [4] which can be seen as a more general and more complex approach. We provide a simpler way which does not need any special kind of dynamic data types and it uses classical partial abstract data types. In GAT, the graphical notation helps the specifier in building its specification. We have a hierarchical approach and the initial algebra existence is ensured as soon as termination is proved. We have a first-order way to proof temporal logic properties, it is uniform with data proofs. Our method is different from Abrial’s proposal for the B method [1]. Abrial’s approach extends the B method to support events and temporal invariants compatible with refinement.

We have done experimentations with the Larch Prover [7]. Let us note that the Larch Prover has some advantages, especially its powerful ordering and rewriting mechanism, but it has also some deficiencies. First it only supports first-order logic, second it does not allow user-defined proof strategies. We also expect to improve the proof methodology and strategies to automate proofs using model checking. Therefore, we expect to use PVS [11] which is more efficient, supports a higher-order logic, a model checker, and user-definable proof strategies. This paper mainly describes the translation of a GAT into PVS. The principle is to define a total data type with generators and predicates. Then a partial data type is obtained using the previous one and the definedness predicate. Several ways to translate axioms are discussed and we point out the interest of the automatic generation of proof obligations. PVS generates proof obligations describing type-checking constraints. These obligations are manually proved because the
powerful PVS type-checking is not decidable. Lastly, our paper also shows some temporal proofs.

To illustrate our approach we consider a part of the vending machine described in [13]. The STS of the CC (Cash Changer) component is described in Figure 1. This is a simple cash changer which accepts coins, one at a time and gets out change. This is a part of a drink distributor, see [13] for a comprehensive description. It only accepts coins of one, two or five Euros and gets out only coins of one Euro. The user gives coins and when the sum is sufficient enough an ok transition may be triggered. The user may cancel its transaction. The getOut transition means to get out coins after receiving the price of the drink on the j variable. Figure 1 is basically a guarded finite state ma-

![Figure 1. The CC STS](image)

chine with some notations to represent operation signature. A transition labelled by an operation name represents the effect of an event occurring on the component. Conditional transitions have a guard as additional label. The data type associated to the CC component is named Changer.

The paper is organised as follows. Section 2 presents the GAT approach and its principles. Section 3 discusses the GAT translation into PVS and gives parts of the PVS changer specification. Section 4 describes some examples of verifications using PVS. Lastly, a conclusion summarises the main points of our PVS translation.

2. GAT Specifications

This Section introduces the concepts of Symbolic Transition System (STS) and Graphical Abstract Data Type (GAT). A more detailed presentation of the GAT approach with its semantics and properties may be found in [14].

A GAT for a component is a STS with some static information and an algebraic specification of a data type. As in process algebra, we distinguish two kinds of GAT components: sequential components and concurrent components. For each GAT component we consider two views: the dynamic view and the functional view. The dynamic view is a STS: a finite set of states and a finite set of labelled transitions. Classic finite transition systems, or Labelled Transition Systems (LTSs) have labels which are closed terms. Unlike LTS, our STS labels are operation calls with variables and guards. A state may represent a set of either finite or infinite objects and a transition collects several state changes. This kind of state machine avoids the state and transition explosion problems and makes dynamic behaviours more readable. The functional view is an algebraic specification of a partial abstract data type which formally described both the operations of the component and its STS. The algebraic description of the STS is done by way of state predicates.

In the GAT process specification we suggest starting from the dynamic view of the components since it is the most abstract view. First, the specifier declares the operations, the conditions and the states of the component. This information is represented graphically using a STS. Second, the semantics of the operations are provided (in the functional view) by an algebraic specification. Instead of writing this algebraic specification from scratch, we propose a guideline. The core of our extracting method is the AG-derivation algorithm [14], which guides the axiom generation using the STS. It provides an operational specifica-

![Figure 2. The GAT Process and Semantics](image)
of STSs is used before generating the axioms. Figure 2 describes an overview of the GAT process and semantics. An algebraic specification is extracted from a STS and its semantics is a partial abstract data type. The STS represents a graphical view of a partial equivalence relation over the data type.

2.1. Partial Abstract Data Type

We consider partial abstract data types, because our STSs need partial operations. We consider initial semantics due to its close relation with proofs and deduction. To gain expressiveness we consider hierarchical presentations of algebraic specifications, with constructors, hidden symbols and definedness predicates. The notations below come from [17]. A signature $\Sigma = (S_0, F)$ is a tuple where $S_0$ is a set of sorts and $F$ a $S_0$-indexed family of function symbols such that $F$ is equipped with a mapping type : $F \rightarrow S_0^* \times S_0$. When there is no ambiguity, we identify $f$ and $\text{type}(f)$. As usual, arity, argument types and range type are defined upon $f$. The type currently defined will be called the Type of Interest and noted $TI$. An internal operation has $TI$ as resulting type. An external operation (or observer) does not have $TI$ as resulting type. A basis internal operation has not any parameter of type $TI$. We also distinguish constructor or generator as the subset of the internal operations sufficient to generate all the value of the data type. We consider a $\text{Spec}_{TI}$ hierarchical specification

$$\text{Spec}_{TI} = \langle \Sigma, HS, E, \text{Cons}, D, P \rangle$$

with respectively signature, hidden signature, axioms, constructors, definedness predicate and primitive part. Partial algebras are algebras where the functions may be partial. Functions are assumed to be strict. A partial algebra is a total algebra such that the interpretation of any term $t$ of sort $s$, in $A$ is defined if and only if $A$ satisfies a definedness formula $D_s(t)$. Notions of homomorphisms and valuations may be defined, note that variable quantifications range over defined values ($D_s(X)$). Here we omit semantic and deduction issues. We consider initial algebra and first-order logic deduction, see [17, 3, 14] for details.

2.2. Symbolic Transition System

The finite state machine formalism is well-known by practitioners. It is well-suited to the description of interactions and controls. One problem with such a formalism is the great number of states and transitions. It often happens if one has a mixed system with both control and data types. We define the notion of finite and symbolic transition system. This notion arises also from the need of a full semantics for languages like LOTOS [16]. See Figure 1 page 2 for an example of STS. Let $St = \{s_i, 1 \leq i \leq n\}$ a set of identifiers called the set of states. A symbolic transition systems is a finite set of state $St$ and a finite set of labelled transition $Tr$. Note also that we allow receipt variables both in guard and in term labels. Variables occurring in the guard and in the term label are not necessarily the same, but in order to simplify we consider the same set of variables in both terms. Symbols and terms occurring in the STS have to be interpreted in the context of the algebraic specification. The term label can be any algebraic term. Without lost of generality we restrict them to functional terms $f(\text{self}, X_1, ..., X_n)$ where $f$ is an operation label. The transitions correspond to internal operations of $TI$ with an interpretation formula based on state predicates. An edge, from state $s_i$ to state $s_j$, is labelled by $[G(\text{self}, X_1, ..., X_n)] f(\text{self}, X_1, ..., X_n)$ if and only if, if $\text{self}$ is in state $s_i$ and $G(\text{self}, X_1, ..., X_n)$ is true then $f(\text{self}, X_1, ..., X_n)$ is a value of state $s_j$. Our notion is more general than the symbolic transition graph defined in [8]. We have more general states (not only tuples of conditions) and we have no restriction on variables occurring on transitions.

2.3. GAT Definition

A Graphic Abstract data Type description is an abstract specification of a data type using a STS ($STS_{TI}$), and a hierarchical specification as in Section 2.1 ($\text{Spec}_{TI}$).

$$\text{GAT}_{TI} = (STS_{TI}, \text{Spec}_{TI})$$

Such a pair defines an associated equivalence relation: $x \approx_{TI} y$ if and only if $x$ and $y$ have the same state. Let $\{P_n\}_{1 \leq n \leq n}$ be a finite set of boolean functions called state predicates. These functions are interpreted as the characteristic functions of the subsets $s_i$.

**Lemma 2.1** $\{P_{s_i}\}_{1 \leq i \leq n}$ verifies the following properties:

- **exclusivity:** $\forall s_i, s_j, s_i \neq s_j \Rightarrow \neg (P_{s_i} \land P_{s_j})$ (1)
- **complementarity:** $D_{TI} = \bigvee_{1 \leq n \leq n} P_n$ (2)

and conversely, if $\{P_{s_i}\}_{1 \leq i \leq n}$ is a set of state predicate which verifies the two above properties then it defines a partial equivalence relation:

$$\text{if } D_{TI}(v) \land D_{TI}(v') \text{ then } v \approx_{TI} v' \Leftrightarrow \exists s_i P_{s_i}(v) \land P_{s_i}(v')$$

In the sequel we use the following notations: $\text{self} : TI$ denotes a variable of type $TI$. $D_{TI}$ is the definedness predicate for $TI$. $P_{s}$, are the state predicates. $\text{precond}_{op}$ is the precondition of the $op$ operation, $G$ will be a guard, * is a tuple of variables, $op_B$ (respectively $op_R$) denotes a basis
internal operation labelling an initial transition (respectively a recursive one labelling a non initial transition). A transition from a source state to a target state will be noted

\[ [G(self,*)] op_{T}(self,*) \]

source \rightarrow target

Note that some of the following definitions are higher-order definitions since sometimes they are defined relatively to a state name or a function name.

We need some auxiliary operations to implement the STS inside the algebraic specification. The formulas for these operations are the same for sequential or concurrent components. All these formulas are generated automatically from the STS description. We consider them as total boolean functions inductively defined on the data type. The definedness predicate is inductively defined by:

\[
D_{T1}(op_{T}(*)) \equiv \text{precond}_{op_{T}}(*)
\]

\[
D_{T1}(op_{T}(self,*)) \equiv \text{precond}_{op_{T}}(self,*) \land D_{T1}(self)
\]

This predicate denotes if a given term of type $T1$ represents a value of the partial algebra or not. The operation preconditions have the from:

\[
\text{precond}_{op_{T}}(*) \equiv \bigvee_{G(*)} G[*]_{target}
\]

\[
\text{precond}_{op_{T}}(self,*) \equiv \bigvee_{P_{source}(self) \land G(self,*)} P[*]_{source} \rightarrow target
\]

The precondition for an observer is defined with the same formulas as in the case of a recursive generator. The last family is the state predicates:

\[
\text{P}_{target}(op_{T}(*)) \equiv \bigvee_{G(*)} G[*]_{target}
\]

\[
\text{P}_{target}(op_{T}(self,*)) \equiv \bigvee_{P_{source}(self) \land G(self,*)} P[*]_{source} \rightarrow target
\]

These definitions capture the requirement for operation strictness.

3. PVS Translation Principles

This Section focuses on the translation of our GAT specifications into the PVS framework. We translate our algebraic specifications in PVS rather than implementing them into PVS. Since we expect to improve user interactions, readability and straight GAT translation are preferable.

3.1. A Brief Introduction to PVS

PVS (the Prototype Verification System) allows to specify and to verify systems using higher-order logic and a sophisticated type system containing predicate subtype. It provides a mechanism for defining abstract data types [12] in a classical way. User-defined proof strategies can be used to enhance the automation and model checking allows to verify temporal properties of finite state systems. PVS has a strong and not decidable type-checking. In PVS Type-

Correctness Conditions (TCC) are proof obligations that must be discharged before a module can be considered well type-checked. Note that PVS does not support hidden operations. However our GAT may use hidden symbols. We adopt the following principle: symbols are always visible and we add formulas expressing lexical scope and visibility rules. Then, in the sequel we avoid the case of hidden symbols. A last remark to note is that the translation of a GAT into PVS may be supported by an automatic process, however it is not yet implemented.

3.2. Translation Choices

The first way to translate a GAT hierarchical presentation is to mimic the Larch Prover translation. That is to define a total data type with a definedness predicate and conditional axioms. Partial first-order logical deduction may be embedded into the total logical framework but two hypotheses have to be realized [3]. The first one is operation strictness, it is ensured by the definedness predicate of Section 2.3. The second one is that care must be taken to be sure that all the terms occurring in axioms and formulas are defined. By construction the terms generated by AG-derivation and occurring into auxiliary operations verify the definedness predicate.

To take advantage of the TCC of PVS we rather use the following methodology. We define a total data type for the GAT with the generators, the predicates (state predicates, preconditions and the definedness predicate) and the guards. This type is suffixed with the total keyword. Once this is done, it is possible to define a subtype of this data type which represents the partial data type associated to the GAT. This definition is based on the definedness predicate and the subtype PVS construction. It is called $T1$. This partial data type supports the other observers of the GAT specification. This has the benefit of simplifying some axiom definitions and moreover to automatically generate TCCs.

3.3. Signature Translation

We first define a PVS data type, the generators of which are the generators of the $Con_{T1}$ set. The name given to
Changer_total : DATATYPE
BEGIN
  newChanger (nat_newChanger : nat) : newChanger?
  ok (changer_ok : Changer_total) : ok?
  give (changer_give : Changer_total, coin : Coin) : give?
  cancel (changer_cancel : Changer_total) : cancel?
  get (changer_get : Changer_total) : get?
  getOut (changer_getOut : Changer_total, val : nat 1to5) : getOut?
END Changer_total

Figure 3. The Changer_total PVS Data Type

this data type is $T_{I\text{total}}$. It represents the type of all the possible paths in the STS, correct or not. For instance, for the Changer module the corresponding PVS data type is given in Figure 3. Let us note that if, for instance, ok is a constructor then changer_ok is the associated destructor and ok? is the recogniser predicate. Then we declare the state predicates, the preconditions and the guards of the set $T_{I\text{total}}$. These operations have to be defined on $T_{I\text{total}}$. The corresponding declarations for the Changer module are given in Figure 4. The last part of the declarations corresponds to the

DChanger : [Changer_total -> bool]
Changer : TYPE =
{ cf : Changer_total | DChanger(cf) }

Figure 5. PVS Declarations for Changer

additional observers. These operations are defined only on the partial type. In a third step, we declare the definedness predicate $D_{I\text{total}}$ and the type of the valid paths of the STS, i.e. the elements of $T_{I\text{total}}$ for which the definedness predicate is true. As explained earlier it is really useful to introduce a refinement of the previous data type with the PVS subtype construction and a subtype predicate [15] as in Figure 5.

3.4. Axiom Translation

Now we translate the algebraic axioms into PVS. We begin with the definedness predicate. The definition of this predicate is given inductively. For the Changer module we have the definition of Figure 6. In Section 2.3 the preconditions are defined from the state predicates and the guards. We strictly follow these definitions. In Figure 7 we give an example of precondition definition for the Changer module. The third set of axioms is the set of the guard definitions. These predicates are total and are true only if the corresponding path is a valid path. The guard definitions of the Changer module are like in Figure 8. The fourth set of axioms corresponds to the definitions of the state predicates. Several approaches can be used to define the state predicates. The first one, given in Figure 9 for the onChange

DChanger_global : AXIOM
DChanger(cf) =
CASES cf OF
  newChanger(i) : true,
  get(cf) : DChanger(cf) AND (PtooMuch(cf) OR Pdelivered(cf) OR Pcanceled(cf)),
  cancel(cf) : DChanger(cf) AND
  (PtooMuch(cf) OR PonChange(cf))
ENDCASES

Figure 6. The Changer Definedness Predicate

money_def : AXIOM
money(cf) = (DChanger(cf) AND
{value(sum(cf)) <= stock(cf)})

Figure 8. Guard Definitions

0-7695-1573-8/02/$17.00 (C) 2002 IEEE
state predicate is close to the algebraic definition. We have one axiom for each form of path. The advantage of this approach is that the rewriting part of the proofs are simpler. The main drawback is that we have a lot of axioms and one may forget one of them. The second approach, given in Figure 10, is an inductive definition. Its main advantage is that

\[
\begin{align*}
\text{PonChange\_new} : & \text{AXIOM } \text{PonChange(newChanger(n))} \\
\text{PonChange\_ok} : & \text{AXIOM } \text{PonChange(ok(cf))} = (\text{PonChange(cf) AND sufficient(cf) AND money(cf)}) \\
\text{PonChange\_give} : & \text{AXIOM } \text{PonChange(give(cf,c1))} = (\text{PonChange(cf) AND NOT(sufficient(cf)))} \\
\text{PonChange\_cancel} : & \text{AXIOM } \text{NOT(PonChange(cancel(cf)))} \\
\text{PonChange\_get} : & \text{AXIOM } \text{PonChange(get(cf))} = (\text{PtoOMuch(cf) OR Pcanceled(cf) OR Pdelivered(cf)}) \\
\text{PonChange\_getOut} : & \text{AXIOM } \text{NOT(PonChange(getOut(cf,i)))}
\end{align*}
\]

Figure 9. onChange: the Axiomatic Approach

the PVS type-checking detects if we forget one case and generates a TCC. The main drawback with such a unique axiom is that the proofs are not readable. There is a last approach, unfortunately it cumulates the two previous drawbacks. We choose the first approach and prove the second one in order to be sure that we do not forget a case. The last

\[
\begin{align*}
\text{OnChange\_global} & : \text{LEMMA} \\
\text{PonChange(ccf)} & = \text{CASES ccf OF} \\
\text{newChanger}(n) & : \text{true,} \\
\text{get(cf)} & : (\text{PtoOMuch(cf) OR Pcanceled(cf) OR Pdelivered(cf),} \\
\text{give(cf,c1)} & : (\text{PonChange(cf) AND NOT(sufficient(cf)))}, \\
\text{ok(cf)} & : (\text{PonChange(cf) AND sufficient(cf) AND money(cf))}, \\
\text{getOut(cf,i)} & : \text{false,} \\
\text{cancel(cf)} & : \text{false} \\
\text{ENDCASES}
\end{align*}
\]

Figure 10. onChange: the Inductive Approach

set of axioms defines the additional observers. We choose to define them with an axiomatic way close to the algebraic axioms. We have one PVS axiom for each case. We cannot use an inductive definition because these operations are defined on the partial data type. The PVS type-checking mechanism is able to generate TCCs. The proof of these TCCs is very useful and allows us to find some errors in the specification. For instance, in a first version of the specification of the toGet observer, the axiom toGet\_getOut was written as follows:

\[
\begin{align*}
\text{toGet\_getOut} : & \text{AXIOM} \\
\text{PonChange(ch) IMPLIES} \\
\text{(toGet(getOut(ch,i)) = built(value(sum(ch)) - i)}
\end{align*}
\]

A TCC cannot be proved and it suggests an error. The right axioms defining this observer are given in Figure 11. As we previously noted, the TCCs reveal many problems

\[
\begin{align*}
\text{toGet\_getOut} & : \text{AXIOM} \\
((\text{PonChange(ch) AND sufficient(ch)) IMPLIES toGet(getOut(ch,i)) = built(value(sum(ch)) - i})
\end{align*}
\]

Figure 11. The Correct toGet\_getOut Axiom in PVS

and are really useful to improve the specification. This is also true with the case analysis of the inductive approach of the observers. On this example 12 TCCs are generated and all of them are manually proved.

4. Early Verifications

One translation is done some specific verifications may be done on the several modules of the vending machine. The two first properties that we can prove are those given in lemma 2.1: the exclusivity and the complementarity. The verification of these properties is important because it allows us to detect specification errors early in the development process.

The exclusivity property for the Changer example is given in Figure 12 and is proved by induction. The complementarity property corresponds to the lemma given in Figure 13 and is also proved by induction.

\[
\begin{align*}
\text{changer\_exclusivity} & : \text{LEMMA} \\
\text{FORALL (cf : Changer)} : \\
\text{NOT(PonChange(cf) AND PtoOMuch(cf)) AND} \\
\text{NOT(PonChange(cf) AND Pcanceled(cf)) AND} \\
\text{NOT(PonChange(cf) AND Pdelivered(cf)) AND} \\
\text{NOT(PtoOMuch(cf) AND Pcanceled(cf)) AND} \\
\text{NOT(PtoOMuch(cf) AND Pdelivered(cf)) AND} \\
\text{NOT(Pcanceled(cf) AND Pdelivered(cf))}
\end{align*}
\]

Figure 12. The Changer Exclusivity Property

Note that with the Larch Prover this proof needs two lemmas, one for each implication. The difficulty comes from the chosen orientation for the rewriting rules of the system.

A third basic property can be proved on each module. In [13] we have shown that the absence of deadlock may be expressed by the formula of Figure 14. This property expresses the fact that there is no deadlock in the STS of the Changer. We proved it by induction.
changer_complementarity : LEMMA
FORALL (cf : Changer) :
(DChanger(cf) =
(FonChange(cf) OR PtooMuch(cf) OR
Pcanceled(cf) OR Pdelivered(cf)))

Figure 13. The Changer Complementarity Property

changer_nodeadlock : LEMMA
FORALL (cf : Changer) :
(PChanger(cf) IMPLIES
(Pgive(cf) OR Pget(cf) OR
Pcancel(cf) OR PgetOut(cf) OR
Pok(cf)))

Figure 14. The Changer No Deadlock Property

We have done other experimentations. The three previous proofs have been successfully done on the other sequential component of this case study. Then there have been also been done on the concurrent composition of these two sequential components: the vending machine. We tried other examples to test the ability of PVS to prove properties. We are able to check for deadlock, even with the presence of conditions. The GTcancel formula of Figure 15 expresses that the finite execution throws a transition labelled by cancel. The formula of Figure 16 denotes that there is no deadlock if we cannot trigger cancel and if the guards sufficient and money are true.

GTcancel : [Changer_free -> bool]
GTcancel_new : AXIOM NOT(GTcancel(newChanger(n))
GTcancel_give : AXIOM
GTcancel_give(ccf,cl) =
(precond_give(ccf) AND GTcancel(ccf))
GTcancel_get : AXIOM
GTcancel_get(ccf) =
(precond_cancel(ccf) AND GTcancel(ccf))
GTcancel_cancel : AXIOM
GTcancel_cancel(ccf) = precond_cancel(ccf)
GTcancel_getOut : AXIOM
GTcancel_getOut(ccf,i) =
(precond_getOut(ccf) AND GTcancel(ccf))
GTcancel_ok : AXIOM
GTcancel_ok(ccf) =
(precond_ok(ccf) AND GTcancel(ccf))

Figure 15. PVS Definition of GTCancel

We may also prove the existence of a deadlock if we disallow the get action (Figure 17).

A last proof (Fig. 18) shows that the value of the amount before an ok action followed by a getOut action is the sum of the price of the drink and the money given out to the user. Note that during this last proof with the Larch Prover we observed a lack in the specification about the price which is bound by 5. This problem was early detected with the TCCs of PVS because they force the specifier to define more precise subtypes.

NogetDeadlock : LEMMA
EXISTS (cf : Changer_free) :
(NOT(GTget(cf)) AND
NOT(precond_give(cf)) AND
NOT(precond_cancel(cf)) AND
NOT(precond_getOut(cf)) AND
NOT(precond_ok(cf)))

Figure 17. a PVS Formula for a Deadlock

Comparing these experimentations with the ones done with the Larch Prover leads to the following conclusions. Of course the Larch Prover user interface is poor. Specifications with PVS and formulas are simpler and shorter than with the Larch Prover. The automatic generation of proof obligations enforces the set of verification and increases the safety. The way to process a proof with the Larch Prover is generally successful with PVS but sometimes it is simpler. PVS integrates arithmetic and logical decision procedures which are useful to mechanise some proofs. The Larch Prover does not have such procedures and it is difficult to process even simple arithmetic formulas. Lastly PVS seems more efficient, some proofs need several hours with the Larch Prover but only a few seconds with PVS.

Nat1to5 : THEORY
BEGIN nat1to5 :
TYPE = { i : nat | i >= 1 AND i <= 5}
CONTAINING 1
END Nat1to5

PropertyValue : LEMMA
FORALL (cf : Changer, price : nat1to5) :
Pdelivered(getOut(ok(cf), price)) IMPLIES
value(toGet(getOut(ok(cf), price)))) + price = value(sum(cf))

Figure 18. a PVS Formula with value and price
5. Conclusion

The GAT method is based on STSs and partial algebraic abstract data types. This method provides assistance, tools and ensure some properties. One important point is the use of STS which allows one to avoid the state and transition explosion problems of usual state automata. Our previous experiments have used the Larch Prover and in this paper we consider that PVS is a better tool for us. In this work we show how to embed a GAT first-order description into PVS. The use of TCCs and case analysis early detects ill-formed definitions. We have also experimented with some proofs already done with the Larch Prover and all of them have been successfully done with PVS. We noted that several proofs are simpler with PVS than with the Larch Prover.

A future direction is to embed the GAT higher-order theory itself into PVS. This will be done throw the implementation of a translation tool and the higher-order logic of PVS will be useful. Another future work is to integrate some automatic strategies especially for temporal logic properties. We have begun to explore two ways: the use of classical model checking and the definition of proof strategies. These two approaches will benefit from the ability of PVS to interact with model checking [10] and to define user proof strategies.

References