Formal Specification and Temporal Proof Techniques for Mixed Systems

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

Abstract

Formal specifications of mixed systems are 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 a fully integrated semantics. This paper presents a proposition trying to cope with this issue: the Graphic Abstract data Type (GAT) approach. GAT is a mixed formalism based on Symbolic Transition Systems (STSs) and algebraic specifications of partial abstract data types. The first part of this paper deals with the specification of a vending machine using the GAT approach. The second part is devoted to prove temporal properties on a GAT specification. Because it is a symbolic system with guards, variables and data values, classical model checking is not sufficient enough, furthermore the specified system is not bound. We rather advocate proofs with a general theorem prover and the use of functional operators expressing temporal properties. We consider that properties are generally twofold, a simple part provable on the STS, considered as a classical finite state machine, and a symbolic part which needs a general theorem prover. We show several examples of properties and proofs.

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. Static aspects deal with the signatures, the types and the relations between types. Functional aspects describe the semantics for operations or explicit some conditions and invariants. Dynamic aspects focus on the so-called dynamic behaviour of systems, it is related to concurrency and communications. 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.

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 [10, 4, 14, 9]. Some proposals separate the process calculi from the data calculi [21], others have a two or multilayered algebraic specification. We have chosen an integrated approach with a single semantic framework based on partial algebras [5, 22].

To specify mixed systems, we proposed the Graphical Abstract Data Type concept [2]: a symbolic transition system coupled with an algebraic specification. In our approach, we distinguish two kinds of components: sequential components and concurrent components. For each component we consider two views: the dynamic view and the functional view. The dynamic view describes the static and the dynamic part of the component, it uses a notion of finite and Symbolic Transition System (STS), for related work see [15, 20, 18]. The functional description is an algebraic specification of a partial abstract data type. We defined a notion of compatibility between a STS and an algebraic specification. Proofs in our context need only one formalism and one environment.

Model checking [7] is a relevant way to prove temporal properties. However it is really suited to finite state systems, whereas we focus here on more general state machines: symbolic transition systems. Other approaches are more relevant to our context [13, 15, 20, 9], but they need either two proof techniques or complex languages mixing temporal logic and data types. In our approach we rather advocate the use of a general theorem prover in conjunction with a STS. We express temporal properties over paths on the STS, and use algebraic operators to define properties on states and transitions or to express past and future. To prove a given formula with data values may need a general theorem prover. But we can derive simpler formulas which are necessary or sufficient conditions to ensure the original formula. These simpler formulas may be proved by classical model checking or graph tools. The STS is useful to automate the formula generations, but also to guide the proofs with the theorem prover. The new approach presented here
has three main characteristics. It proposes a uniform way to manage and to prove data properties and temporal logic formulas. Second, it is based on first-order predicate logic and it handles any kinds of data type, either finite or unbound. Finally, it provides the specifier with the abstraction and readability of state-transition systems.

The paper is organised as follows. Section 2 gives an overview of the informal description of our case study: the vending machine. Section 3 presents the GAT approach and its application to the vending machine. Section 4 discusses verification, namely model checking and theorem proving. Section 5 describes some principles to prove dynamic properties with a theorem prover on a GAT specification. Last, Section 6 shows some proofs about the vending machine.

2. The Vending Machine Case Study

In order to specify a system, a (formal) language is required but also an adequate method. In the following specification we do not describe the specification process but only the resulting specification. We consider that a preliminary analysis was done and produced a system architecture. This decomposition can be obtained using some already known methods. For instance, methods for LOTOS [21, 18], are relevant here.

2.1. Informal Description

We deal with a vending machine (a French one) which accepts coins, gets out change and delivers a drink. To simplify, it only accepts coins of one, two or five Francs and gets out only coins of one Franc. The user gives coins, one at a time, and when the sum is sufficient enough he may choose a drink. If this drink is in the stock then the user gets it, else he has to do another choice. The vending machine cannot allow choices if it does not have enough money to get out change to the user. The price of the different kinds of drink are not supposed to be the same but the maximum cost of one drink is assumed to be five Francs.

Figure 1. The Vending Machine Architecture

gives too much than required, or get out the change for the difference between the given sum and the price of the drink. CANCEL is used to interrupt the transaction, and is a ternary synchronization. CHOOSE allows one to choose a drink. DRINK delivers the chosen drink to the user. OK denotes that the sum is sufficient to deliver a drink. GETOUT means that the DD component returns the cost of the chosen drink to the CC component.

3. A GAT Specification

In this Section we provide a formal specification of the VM machine using the GAT approach, a comprehensive specification and a LOTOS one may be found in [19]. A more detailed presentation of the GAT approach for sequential components with its semantics and properties may be found in [1].

A GAT for a component is a STS with some static informations and an algebraic specification of a data type. 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 state and a finite set of labelled transition. Classic finite transition system, or Labelled Transition System (LTS) have labels which are closed terms. Unlike LTS, our STS labels are operation calls with variables and guards. This concept is related to machines where states and transitions are not necessarily unique objects. 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 [5].

In the GAT process specification we suggest to start 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. These informations are represented graphically using a STS. Second, the semantics of the operations is provided (in the functional view) by an algebraic specification. Instead of writing this algebraic specification from scratch, we propose some guidelines. The core of our extracting method is the AG-derivation algorithm [2], which guides the axiom generation using an STS. It provides an operational specification style where axioms may be transformed into left-to-right conditional rewriting rules. In case of concurrent and communicating components, the synchronized product of STSs [2] is used before generating the axioms. Figure 2 describes an overview of the GAT 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.

3.1. The CC Component

The STS of the CC component is described in Figure 3. This is basically a guarded finite state machine 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. Its algebraic specification has a signature and positive conditional axioms. The STS describes the signature following the notations explained below. An internal operation has the Type of Interest (TI) as resulting type, solid arrows denote internal operations (give, ok, getOut, newChanger, cancel). A basic internal operation has no any parameter of type TI, it is depicted with a dashed arrow (newChanger). The set of generator is a set of operations which defines the inductive structure of the TI terms. In the different examples we choose the set of internal operation to be the set of generator. An external operation (or observer) does not have TI as resulting type, such operations are drawn with dotted arrows (toGet). Additionally to the operations described in

![Figure 2. The GAT Semantics](image)

![Figure 3. The CC STS](image)
For each state we try to give a right-hand side conclusion term. If it is not possible we replace the \( ch \) variable by the generator calls reaching this state. The conditional part changes according to this replacement and the specifier must provide either the right-hand side terms or the process continue. The derivation process stops either with a non recursive generator or in a state already visited. With the `tooMuch` state we get the two following derivations.

\[
\text{tooMuch state}
\]
\[
\text{tooMuch}(ch) \Rightarrow \text{toGet}(ch) = ?
\]

Finally we get the following axioms for this operation.

\[
\text{assert } \%	ext{ to declare axioms}
\]
\[
\text{the tooMuch state}
\]
\[
(\text{sufficient}(ch) /\ \text{sufficient}(ch)) \Rightarrow
\text{toGet}(\text{give}(ch, c1)) = ?
\]
\[
\text{tooMuch}(ch) \Rightarrow
\text{toGet}(\text{give}(ch, c1)) = ?
\]

3.2. The DD Component

The same process is achieved for the other sequential component and we get the STS (see Fig. 4) and an algebraic specification for the associated `Distributor` data type.

3.3. The Machine Component

Once the CC and DD components are specified, we build the GAT for the whole VM machine (Fig. 5). First, we build the synchronous product [3] of the two previous diagrams, this gives us the global dynamic behaviour of the VM machine. The list of synchronous actions is: \((\text{OK, OK}), (\text{GETOUT, GETOUT}), (\text{CANCEL, CANCEL}), (\text{GIVE, -}), (\text{GET, -}), (-, \text{CHOOSE}), (-, \text{DRINK})\) where - denotes no action on the corresponding component. The synchronous product can be computed with the help of tools [6]. The `Machine` data type associated to the VM component is based on the product of the component data types `Changer` and `Distributor`. Two selectors, the left and right operations, are defined for the `Machine` to retrieve these component data types. We associate an operation name to each pair of action in the product. For example the \((\text{OK, OK})\) synchronous action is named `ok`. The shorthand for guards are: \( S \) for `sufficient(left(m))`, \( M \) for `money(left(m))`, \( IS \) for `isThereDrink(right(m))`. Since the product of STS is still a STS we can apply the extracting principle to get an algebraic specification for the `Machine` data type.

This seems a bit hard to write by hand but several parts are automatically generated by tools [6]. The computation of the algebraic specification is partly automatic, which is an advantage for non expert designers. The signature, the preconditions and the state predicates axioms are automatically computed from the STS. As we previously saw it, the AG-derivation mechanism provides help to generate axioms of the data type. The resulting specifications have also interesting properties. We may implement the specification, transforming axioms into left-to-right rewriting rules. The properties of consistency, sufficient completeness and existence of initial algebra are achieved without difficulty [2].

4. Verification

There are several ways to verify a system. The most general way is to use a theorem prover. Another way, devoted to particular properties (namely temporal properties) is model checking. It uses a temporal logic and efficient algorithms to prove temporal properties. Temporal logics are logics augmented with temporal operators.
4.1. Symbolic Model Checking

Model checking is a technique to verify automatically temporal properties of finite state systems. Tool examples are CADP, MEC, VIS, see [3, 7] for more details. Model checking is useful to quickly prove deadlock or other related properties.

The state explosion problem of model checking can be limited by using BDD coding. This technique is called symbolic model checking, although it does not address the worst-case complexity of the problem. In practice it is useful and allows the verification of very big systems with more than one million of states. In our example we must bound the data types or the labelled transition system explodes. Let us consider that a user may have three coins (1F, 2F, and 5F), the stock of the changer has 100 coins of 1F and the stock of drinks has four kinds of 10 bottles each. With these decent-size hypothesis we estimate that the resulting STS would have about $10^6$ states and $10^8$ transitions and may reach the limit of current model checkers. But another problem is the lack of abstraction and readability. With a look at Figure 5 we can see that we only have eleven symbolic states and thirty four symbolic transitions. In our approach we control the size of the STS and we can handle unbound data values. These are two great advantages of the GAT specification with symbolic transition systems.

We have considered that the model checking approach is not suitable to our example. However it may be helpful to use classical (or symbolic) model checking to quickly prove some properties of the finite system forgetting guards and variables. For example, to known if the finite state machine has a deadlock or if there are transitions starting from a given state, and so on.

4.2. Model checking with variables

There are two main points which are generally not covered by classical model checking and temporal logic. The first is the use of guards, variables and full data types. The second is that we do not deal generally with a bound LTS.

There are several different approaches which introduce variables and guards in model checking. Amongst them the μCRL and (full) LOTOS languages are relevant to our problem. LOTOS [21] is an approach which permits to specify concurrent processes with data types. The standard semantics of LOTOS does not take into account full data types but only ground expressions. A symbolic semantics for Full LOTOS is a way to overcome this limitation. [20, 16] use model checking coupled with a theorem prover. Dynamic properties are proved on the STS with the help of an oracle on data types. A similar approach for CSP is [15]. Another way [13], proposes an extension of the μ-calculus with data types and variables. We have not yet investigated the comparison in depth, but clearly our approach is more suited to engineering practice (first-order logic framework). We also have more readable state-transitions systems and a more powerful framework.

Both our approach and abstract dynamic data types (AD-DT) [9] use partial abstract data type. In AD-DT the specification is twofold: algebraic axioms and a ternary predicate for the transition system. Our approach is simpler because we have only one layer in the algebraic specification. Another important difference is that we use STS a priori to build the algebraic specification. In AD-DT, the STS is built a posteriori from the algebraic specification. However we do not have yet a temporal logic as in [9].

4.3. Theorem Proving

In our approach we suggest to use a general theorem prover with the help of a model checker or a graph tool. Examples of such provers are PVS [17], HOL [12], and the Larch Prover. We propose the opposite way of previous quoted works that use an oracle: the main technique uses a theorem prover, model checkers and graph algorithms may help to process proofs with variables and guards. Until now we do not know other similar approaches.

In our approach we have first-order formulas which express general properties and we rather focus on dynamic properties with some data values. Because this is a general algebraic approach we are able to define new operations when this is needed. One important fact is that the axioms of these operations may be generated automatically from the GAT. One may also define more sophisticated operators, for instance an operator computing the number of steps between two states or the number of occurrence of a given action or the duration of a sequence of actions.
We use the Larch Prover which is a proof assistant based on first order predicate calculus with equality. It has a set of proof tools, mainly they are: rewriting, critical pair computation, Knuth-Bendix completion procedure, proof by induction, proof by contradiction, and proof by case. It allows one to define formulas with quantifiers (\(\land, \lor, \forall\)) and provides some commands to eliminate quantifiers during proofs. Boolean logic operators are \(\land\) for and, \(\lor\) for or, \(\Rightarrow\) for implication, and \(\equiv\) for equality. The main commands are: prove to start a proof, resume by to try a proof technique, and qed to check if the proof is finished.

5. GAT Proofs Principles

Note that our approach is mainly relevant for systems with complex data types. Since it is a general approach, it is not completely automatic. However we may sometimes automatically prove or disprove the absence of deadlock. Efficiency, of course is difficult to consider because we have not the same LTS size and since generally proofs are not automatic.

5.1. Hypotheses

In the sequel we use the following notations: \(sel f : TI\) denotes a variable of type \(TI\), \(D_{TI}\) is the definedness predicate, \(pred_s\) are state predicates, \(precond_{op}\) is the precondition of the \(op\) operation, \(G\) will be a guard, \(*\) is a tuple of variables, \(op_B\) (respectively \(op_R\)) denotes a basic internal operation (respectively a recursive one).

There are transitions in the STS without guards, these guards are implicitly equal to true. Other transitions have explicit guards. We consider finitely generated values, i.e. every values can be denoted by a finite sequence of generators [22]. The following property states that each state has at least one finitely generated value.

**Definition 5.1 (State Compactness)** \(\forall s \in S, \exists sel f : TI, D_{TI}(sel f) \land pred_s(sel f)\), where \(S\) is the set of states of the STS.

One may check the state compactness using a similar technique as in Section 6.2 and this has been done for the three GATs of the machine (the two components and the whole machine).

5.2. GAT Operations

We present in this Section a summary of some (family) of operations generated by the GAT method. The formulas are the same for sequential or concurrent components (except for the state predicates). The first family is the state predicates, we consider them as total predicates inductively defined on the STS by:

\[
pred_{\text{target}}(op_B(*)) \equiv \bigvee_{G(*)} G(*)
\]

\[
pred_{\text{target}}(op_R(sel f, *)) \equiv \bigvee_{[G] \in \text{op}_{op_{B}}(sel f, *)} \text{pred}_{\text{source}}(sel f) \land G(sel f, *)
\]

(1)

For the concurrent case, a state name for the product is built from the component state names. Let \(l\) and \(r\) be two component states, a state name of the product is a pair \((l, r)\). The state predicate is simply defined as:

\[
pred_{(l, r)}(sel f) \equiv \text{pred}_l(\text{left}(sel f)) \land \text{pred}_r(\text{right}(sel f))
\]

(2)

The definedness predicate is inductively defined by:

\[
D_{TI}(op_B(*)) \equiv \text{precond}_{op_{B}}(*)
\]

\[
D_{TI}(op_R(sel f, *)) \equiv \text{precond}_{op_{B}}(sel f, *) \land D_{TI}(sel f)
\]

(3)

This predicate denotes if a given term represents or not a value of the partial algebra. Operation preconditions have the form:

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

\[
\text{precond}_{op_{B}}(sel f, *) \equiv \bigvee_{[G] \in \text{op}_{op_{B}}(sel f, *)} \text{pred}_{\text{source}}(sel f) \land G(sel f, *)
\]

(4)

There are some properties which link the previous definitions: The state predicates are exclusive and complementary because they define a partial equivalence relation on terms.

\[
\forall sel f : TI, D_{TI}(sel f) = \bigvee_{s \in S} \text{pred}_s(sel f)
\]

(5)

\[
\forall sel f : TI, \forall s, s' \in S, s \neq s' \implies \neg (\text{pred}_s(sel f) \land \text{pred}_{s'}(sel f))
\]

(6)

To prove exclusivity and complementarity (properties 5 and 6) is a first mean to check some problems in the algebraic specification. These properties have been proved for the three GATs of the VM case study.

5.3. Simple and Symbolic Properties of STS

An execution path for the machine is denoted by a variable of type \(TI\). A property will be called STS simple if it does not contain other variables than path variables, else the property is called symbolic. As an example, formula 13 of
Section 5.5 is symbolic whereas formula 14 of Section 6.1 is STS simple. The use of a model checker or a graph analysis may prove or disprove a STS simple formula. A symbolic property requires to evaluate variables and guards. Given one symbolic property \( P \) we can get several simple properties from it. Two ways seems relevant here:

- To be optimistic \( (F_{opti}) \): we consider that all the explicit guards are true.

- To be pessimistic \( (F_{pessi}) \): we consider that all the explicit guards are false.

These two transformations are also available to STSs. Let \( Sts \) be a STS:

- \( Stsopti \) is the \( Sts \) considering that all guards are set to true.

- \( Stspessi \) is the \( Sts \) without transition labelled by explicit guards. However, to preserve the compactness property, we must keep all the states. If a state has not any reaching transition, it is considered as initial (our STSs may have several initial states).

From a \( F \) formula and a \( Sts \) we can generate the optimistic and pessimistic associated counterparts. Of course \( Sts_{opti} \) and \( Sts_{pessi} \) may be viewed as labelled transition systems and \( F_{opti} \) and \( F_{pessi} \) are simple formulas which may be proved by model checking, graph tools or other proof techniques. We may define logical relations linking the different formulas, thus we may expect to automatically prove the simple formulas and then to get a condition for either \( F \) or not \( F \).

### 5.4. Temporal Operator Examples

We present in this Section some temporal operators which are used in proofs of the next Section. An operator is a set of similar algebraic definitions. We provide below generic definitions based on the generators.

The next operator allows one to know if two states are consecutive in the system. The general form of next for a system is:

\[
\begin{align*}
\text{next}(s & \text{elf}, \text{op}_B(*) \equiv false \\
\text{next}(s & \text{elf}, \text{op}_R(\text{sel}f, *)) \equiv D_{TI}(\text{op}_R(\text{sel}f, *))
\end{align*}
\]  

An example with the VM machine in LP is:

```c
next(m, newMachine(i, ld1));
next(m, cancel(m)) = Dmachine(cancel(m));
... 
next(m, getout(m)) = Dmachine(getout(m));
```

The prefix binary operator is true if the first path is a prefix of the second one, it allows branching logical time. The general form for prefix is:

\[
\begin{align*}
\text{prefix}(s & \text{elf}, \text{op}_B(*) \equiv (s & \text{elf} = \text{op}_B(*)) \\
\text{prefix}(s & \text{elf}, \text{op}_R(\text{sel}f, *)) \equiv \text{precond}_{op}(sel'f, *) \\
\text{prefix}(s & \text{elf}, sel'f) \equiv \text{precond}_{op}(sel'f, *) \\
\text{prefix}(sel'f, sel'f) \equiv (sel'f \equiv \text{op}_R(\text{sel}f, *))
\end{align*}
\]  

There is one prefix operation for each GAT. It defines a partial order on execution paths. It also has a non reflexive variant. If prefix \((\text{path1}, \text{path2}) \equiv true\), then \text{path1} is a past of \text{path2}. One may also think \text{path2} as a future of \text{path1}.

The endedWith operator is the symmetric, for a transition, of the state predicate. It is true if a path terminates with a given action. The general form of endedWith for a \( la \) operation label is:

\[
\begin{align*}
\text{endedWith}(la(\text{op}_B(*)) \equiv D_{TI}(\text{op}_B(*)) \land (la = \text{op}_B) \\
\text{endedWith}(la(\text{op}_R(\text{sel}f, *)) \equiv D_{TI}(\text{op}_R(\text{sel}f, *)) \land (la = \text{op}_R)
\end{align*}
\]  

The GT operators are predicates which are true if a given path goes through a given state or a given transition. The general form of GT for a state \( s \) is:

\[
\begin{align*}
\text{GT}\ s(\text{op}_B(*)) \equiv D_{TI}(\text{op}_B(*)) \land \text{pred}_s(\text{op}_B(*)) \\
\text{GT}\ s(\text{op}_R(\text{sel}f, *)) \equiv D_{TI}(\text{op}_R(\text{sel}f, *)) \land \\
\quad (\text{pred}_s(\text{op}_R(\text{sel}f, *)) \lor \text{GT}\ s(\text{sel}f))
\end{align*}
\]  

The general form of GT for a \( la \) transition label is:

\[
\begin{align*}
\text{GT}\ la(\text{op}_B(*)) \equiv D_{TI}(\text{op}_B(*)) \land (la = \text{op}_B) \\
\text{GT}\ la(\text{op}_R(\text{sel}f, *)) \equiv D_{TI}(\text{op}_R(\text{sel}f, *)) \land \\
\quad ((la = \text{op}_R) \lor \text{GT}\ la(\text{sel}f))
\end{align*}
\]  

These operators imply the definedness of the path \((i.e. \text{GT}^*(\text{sel}f) \Rightarrow D_{TI}(\text{sel}f))\).

### 5.5. Links with Temporal Logic

We provide here a brief comparison related to temporal logic, namely CTL [8].

- A finite execution sequence is denoted by a variable like \text{sel}f : TI which verifies the \( D_{TI} \) definedness predicate.

- The system is in a given state \( s \in S \) if and only if \( \text{pred}_s(\text{sel}f) \) is true.

- The precondition of an operation \( \text{precond}_{op} \) means that the corresponding action is possible or not.
Interpretation of some temporal logical operators may be 

\[ AG \Phi \triangleq D_{TI}(sel f) \implies \Phi(set f) \]
\[ EX \Psi \triangleq \exists sel f \ ': TI, D_{TI}(sel f) \implies next(set f, sel f') \wedge \Psi(set f') \]  

(12)

The absence of deadlock may be expressed by \( AG \ EX \ true \) in CTL. With our formalism, considering the different inductive cases for \( sel f' \) and the next definition (Formula 7) we transform the CTL formula into:

\[ \forall sel f : TI, D_{TI}(sel f) \implies \bigvee_{op_n} (\exists s, precond_{op_n}(sel f, s)) \]  

(13)

Future work must be done to formalize the translation of STS formulas into CTL formulas.

6. Proofs for the Vending Machine

Our experiments have been done with the Larch Prover [11] but more up to date or powerful tools may also be successful. We assume that the algebraic system is oriented into left-to-right rewriting rules. One advantage of GAT is that this orientation may be easily obtained [1].

6.1. Deadlocks

Let \( noDeadlock \) (respectively \( deadlock \)) be the formula denoting the absence (respectively the presence) of deadlock. Our analysis (suggested in Section 5.3) shows that: \( Deadlock_{opti} \implies Deadlock \) and \( noDeadlock_{presi} \implies noDeadlock \). The proof that a GAT has no deadlock may be expressed by the \( noDeadlock \) formula (13). Note that in our example we have no extra conditions on emitted and received objects (coins and bottles). Here we do not have any existential variable, therefore we only have to prove the formula:

\[ \forall sel f : TI, D_{TI}(sel f) \implies \bigvee_{op_n} precond_{op_n}(sel f) \]  

(14)

We have done this proof for our vending machine example, see [19]. One can see in the Figure 5 that there is always one total transition starting from each state, therefore \( noDeadlock \) is obviously true. A formal and automated approach to do that is to consider the pessimistic hypothesis: \( noDeadlock_{presi} \) is true.

A more interesting property, from a user point of view, will be to prove that avoiding \( cancel \), assuming sufficient money and bottles in the stocks and if the user gives sufficient money, then the system has also no deadlock. In this case we are interested by a subsystem of the global system. We avoid the transitions labelled by \( cancel \), and we put additional conditions. We must now prove the following \( noDeadSub \) formula:

\[
prove (D_{machine}(m) \land \neg GTcancel(self) \land \\
\neg cancel(self) \land (\neg GTcancel(self) \land \\
\neg cancel(self)) => \\
\neg GTcancel(self) \land \neg cancel(self)) \]

In this case the \( noDeadSub_{presi} \) is false and carries no useful information, hence the \( noDeadSub \) property must be proved with the theorem prover. This proof was done with a similar script (but a bit simpler) than for the previous \( noDeadlock \) formula. Below is the LP script of this proof:

\[
resume by induction \quad \% basic case \\
resume by => \quad \% ok \\
resume by => \quad \% getout \\
resume by => \quad \% drink \\
resume by case deliveredhere(mc), \\
\quad onChangewhere(mc), tooMuchthere(mc) \\
resume by => \quad \% cancel \\
resume by => \quad \% get \\
resume by case ... \\
resume by => \quad \% give \\
resume by case ... \\
resume by => \quad \% choose \\
resume by case ... \\
qed \\
\]

The \( resume \) command tries a proof by induction on the \( m \) variable. The basis case is proved and then we try a proof by implication (\( resume by => \) to reduce the \( => \) of the formula). It is successful with the \( ok \) and \( getout \) generators. But it is not enough for \( drink \) and it requires a proof by case on the precondition of this generator (the \( deliveredhere, onChangewhere, and tooMuchthere \) states). The rest of the proof is similar. It may seem that it is a difficult proof, but in fact the different cases are exactly the operation precondition. The proofs we have done suggest that proof strategies are possible with the GAT approach.

6.2. Existential Property

One often wants to prove the existence of a path with a given property. This is sometimes difficult, especially with LP. However the STS may be used to get paths which have this property. Moreover we can automatically generate from the STS the lemmas to be verified. Here is an example with a property which asserts that there is a path which reaches the \( onChangewhere \) state and goes through the \( onChangewhere \) state.

\[
declare operator mcte : -> Machine \\
assert D_{machine}(mcte) \land onChangewhere(mcte) \\
% \\
prove \exists m ((D_{machine}(m) \land onChangewhere(m))
\]

\[\]
In the previous proof we define a mcte constant to eliminate the existential quantifier following the Skolem principle implemented by the specialization proof technique. Note that we may provide additional informations about variables, either as values (for example newCoin(5)) or as conditions. For example, the following script describes a proof about such a formula. It asserts the existence of a path which contains a transition [S] give from onchangethere to toomuchthere.

prove \E m \E m1 \E c
((Dmachine(m) \&\& deliveredthere(m1) \&\& prefix(m2, m1) \&\& endedWithOk(m2))) => endedWithDrink(m)
resume by induction
resume by case deliveredthere(mc),
onChangethere(mc), tooMuchthere(mc)
qed

6.3. A Proof with GT

We want to prove that passing through the deliverthere state is mandatory to drink. Precisely we prove “if an execution ends with the drink action then the machine went through the deliverthere state”.

endedWithDrink(m) => GTdeliverthere(m)

A graph analysis of the STS shows that there are several paths which must verify the above property. One may note that such paths go through one of the states: deliverthere, onchangethere or toomuchthere. This suggests to prove the two following lemmas, see [19] for the complete proofs:

\% lemma 1
Dmachine(m) \&\& deliveredthere(m) => GTdeliverthere(m)
\% lemma 2
Dmachine(m) \&\& (onchangethere(m) \&\& toomuchthere(m)) => GTdeliverthere(m)

The last lemma collects two cases in the same context because of their mutual dependency (cycle in the STS). Finally we prove the expected property:

prove endedWithDrink(m) => GTdeliverthere(m)
resume by induction
resume by =>
resume by case deliveredthere(mc), onChangethere(mc), tooMuchthere(mc)
qed

From now on we have done more than thirty proofs on this case study. We proved the 5, 6, and 13 formulas for the three components. We also proved several more complex properties like “if a path ends with a drink action we went through an ok action and then we reached the deliveredthere state”.

We have also some results about proof strategies, for example to proof or disprove deadlock when there is no guard or to prove the 6 property in the general case.

7. Conclusion

In this paper we present a way to algebraically specify concurrent systems and then verify temporal properties. Our method uses the concept of GAT which 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. It also keeps an understandable complexity, this is important to users for writing formulas and for proving them. Another interesting point, since the STSs have rather a small size, is that usual graph or automaton algorithms are efficient enough to compute simple results.

In this paper, we illustrate how to write and to prove temporal properties. We recall that we only must compare things which are comparable and, of course, our STSs are not finite state machine acceptable for classic model checkers even for symbolic ones. The comparison is more pertinent with approaches like [20, 15, 16, 13]. We loose the automation but we only have the need for one well-known logic formalism and proof framework. We do not have yet a full temporal logic. However the readability of temporal logic with data values does not seem obvious. Our approach provides a uniform way to write and to proof both data formulas and temporal logic formulas. It is clear that proofs may be partially automated, we have presented some examples about deadlock. One may note that the STS is also useful to guide the proofs, especially existential proofs.

Here we have presented a first approach with some examples of operators and proofs. This first and informal approach will be improved by a complete set of operators and the formal links with well-known temporal logics. We also expect to improve the proof methodology and strategies to automate proofs using both graph algorithms and model checking. Finally, let us note that LP has some advantages, especially the use of rewriting rules, but it has also some lacks. Therefore, we expect to use PVS [17] which is more...
efficient, supports a higher-order logic, a model checker, and user-definable proof strategies.

References


