Describing and Analyzing Behaviours over Tabular Specifications using (Dyn)Alloy

Nazareno M. Aguirre[1], Marcelo F. Frias[2], Mariano M. Moscato[2], Thomas S. E. Maibaum[3], and Alan Wassyng[3]

[1] Department of Computer Science, FCEFQyN, Universidad Nacional de Rio Cuarto and CONICET, Argentina, naguirre@dc.exa.unrc.edu.ar
[2] Department of Computer Science, FCEyN, Universidad de Buenos Aires and CONICET, Argentina, {mfrias,mmoscato}@dc.uba.ar
[3] Department of Computing and Software, McMaster University, Canada, tom@maibaum.org, wassyng@mcmaster.ca

Abstract. We propose complementing tabular notations used in requirements specifications, such as those used in the SCR method, with a formalism for describing specific, useful, subclasses of *computations*, i.e., particular combinations of the atomic transitions specified within tables. This provides the specifier with the ability of *driving* the execution of transitions specified by tables, without the onerous burden of having to introduce modifications into the tabular expressions; thus, it avoids the problem of modifying the object of analysis, which would make the analysis indirect and potentially confusing. This is useful for a number of activities, such as defining test harnesses for tables, and concentrating the analyses on particular, interesting, subsets of computations. Unlike previous approaches, ours allows for the description of a wider class of combinations of the transitions defined by tables, by means of a rich operational language. This language is an extension of the Alloy language, called DynAlloy, whose notation is inspired by that of dynamic logic.
The use of DynAlloy enables us to provide an extra mechanism for the analysis of tabular specifications, based on SAT solving. We will illustrate this and the features of our approach via an example based on a known tabular specification of a simple autopilot system.

1 Introduction

Tabular notations, originally used to document requirements by D. Parnas and others [9], have proved to be a useful means for concisely describing expressions characterizing complex requirements. Indeed, tables have been successfully incorporated into various formalisms for requirements specification, most notably those reported in [12, 7]. The central use of tables in the description of software requirements is as a way of organizing formulas that specify the relations that the system must maintain with the environment. Since these formulas would be large and complex in their traditionally linear notation, their division into well distinguished smaller formulas that are easier to follow, provided by the tabular notation, has great advantages. A tabular specification then consists of a collection of tables, which combined specify a relation R, characterizing the intended behaviour of the system. There exist different classes of tables, but essentially all are descriptions of relations of some form in terms of *guards* and *result values*. The whole specified system is then typically composed of a disjunction of these guarded expressions, describing, intuitively, all transitions.

We are concerned with complementing tabular notations with a way of prescribing specific combinations of transitions defined in tabular descriptions. As we will argue later on, this enables the specifier to *drive* the execution of transitions defined by tables, which is useful for defining test harnesses for tables, and concentrating the analysis activities on particular execution scenarios, namely those corresponding to the prescribed combinations.

Contributions of this paper. The contributions of this paper are twofold. First, we propose a notation for prescribing subsets of the set of all possible executions of a tabular specification. The notation is very expressive, based on an operational language called DynAlloy [4], an extension of the Alloy specification language [10]. This has as an advantage that the specifier can concentrate the analyses on the particular sets of runs he is interested in with a potentially great impact on the efficiency and effectiveness of the analyses. The proposed notation enables the specifier to describe sets of executions by means of *programs* referring to the tabular descriptions, without the need to introduce modifications in the tables. These descriptions are written in a language accessible to the specifier familiar with tabular descriptions.

Second, we provide an additional analysis mechanism for tabular specifications, based on SAT solving, and supporting the above mentioned notation for prescribing sets of executions. This analysis mechanism is based on a translation of the tabular specifications into Alloy and DynAlloy, and the use of the Alloy and DynAlloy Analyzers for performing SAT based analysis.

Related Work. The described tabular notations, and in particular the tabular expressions used in the SCR (Software Cost Reduction) method [7], have associated tool support, which provide different kinds of analysis, ranging from simple syntax checking to theorem proving and model checking of properties [3, 8, 13]. However, most analyses we are aware of apply to the whole set of behaviours associated with tables; more precisely, most techniques for analyzing properties of behaviours are concerned with the *global* set of “atomic transitions”, described in the tables. Generally, there is a lack of a notation for describing particular combinations of these atomic transitions or tables. An exception to this is the case of the simulator in the SCR toolset [8]. The simulator allows the developer to load specific scenarios and check whether certain associated assertions are violated or not in the particular executions described by the scenarios. Also, in an approach described in [5] and defined for testing purposes in the context of SCR [7], *modes* (essentially classes of states) are exploited as a means for singling out a proper subset of all possible transition sequences allowed by a tabular specification. These approaches have some limitations. The use of the first alternative becomes impractical if the set of execution scenarios one is interested in is large, since each execution scenario needs to be individually described. On the other hand, the second alternative allows for the description (and analysis) of large proper subsets of the set of executions associated with a tabular description, but all these executions are similarly obtained, essentially by considering all the execution sequences that "go through" (or, more precisely, "end up at") a given set of modes. This is insufficient if one is interested in more sophisticated execution sequences, resulting from table sequencings not obtainable by "filtering" executions according to some of the existing modes. Of course, one might decide to include new modes and mode classes to enforce these particular sequencings of the transitions modelled by tables; however, this last alternative is, in our opinion, unsuitable, since it would require altering the tables and introducing new modes and mode classes to enforce the sequencing; clearly, this is inappropriate, and potentially dangerous, if the sequencing is not really part of the requirements, but particular behaviours the modeller wants to analyze.

2 An Example of Tabular Specifications

In order to illustrate how transitions are typically specified by tables, we will use the SCR approach to requirements specification. We will describe the notation via an SCR specification, given in [2], of the requirements for a simple autopilot system. This will also serve us as a case study for illustrating our proposal.

In the SCR methodology, tables are used for describing the relationship that the system should induce between monitored and controlled variables. In order to describe this relationship, SCR uses *events*, *conditions*, *mode classes* and *terms*. Events occur when changes in the variables observed by the system take place (these variables include monitored and controlled variables, as well as modes and terms), and conditions are logical expressions referring to these variables. Modes represent classes of states of the system (whose corresponding partition is called a mode class), and terms are functions on the variables of the specification.

The autopilot needs to monitor three environment variables, the aircraft's altitude, flight path angle and calibrated air speed, represented by monitored variables `mALTactual`, `mFPAactual` and `mCASactual`, respectively. It also monitors the status of some elements in the autopilot's control panel, which are four switches, represented also by monitored variables `mALTsw`, `mATTsw`, `mCASsw` and `mFPAsw`, and three knobs for changing the desired altitude, flight path angle and calibrated air speed (these values are represented by monitored variables `mALTdesired`, `mFPAdesired` and `mCASdesired`, respectively). The system has to control three displays, which show either the actual or desired altitude, flight path angle and calibrated air speed (displays are represented by controlled variables `cALTdisplay`, `cFPAdisplay` and `cCASdisplay`), depending on the state of the system. The four switches allow the pilot to activate the modes `ATTmode`, `ALTmode`, `FPAmode` and `CASmode` of the system. The displays usually show the current altitude, flight path angle and calibrated air speed, unless the pilot changes one of these desired values (i.e., "preselects" a value) and activates the corresponding mode. In this case, the display will show the desired value instead of the actual one. Each display will show the corresponding current value (instead of the desired value) when the corresponding mode is manually disengaged, or when the desired value is reached. Modes are engaged/disengaged by setting the corresponding switches, although the system cannot be engaged in more than one of the modes `ALTmode` and `FPAmode`, so entering one of these should disengage the previous mode. There is an extra mode, the attitude control wheel steering, in which the system is set when neither `ALTmode` nor `FPAmode` are engaged. The `CASmode` can be engaged independently of the other modes, at any time.

When the pilot attempts to engage the system into the `ALTmode`, setting the desired altitude to one that is more that 1200 feet above the current altitude, the system will not engage directly in the `ALTmode`; in this situation, the system will switch to the mode `FPAmode`, in an “armed” mode. Then the system will require the pilot to enter a flight path angle (that to follow until the aircraft gets within 1200 feet away from the desired altitude), after which the system will move to an “unarmed” mode. Once the aircraft reaches the point where it is less than 1200 feet from the desired altitude, it engages the mode `ALTmode`. When a mode other than `CASmode` is engaged, the other preselected displays return to show the current value (instead of the desired one).

In the formalization of the autopilot system’s requirements we are reproducing here, the `FPAmode` is splitted into two different modes, `FPAarmed` and `FPAunarmed`, to differentiate the cases in which the `FPAmode` is *armed* (waiting for the flight path angle to be set after the `mALTsw` was switched on at an altitude lower than 1200 feet below the desired altitude) and the case in which the `FPAmode` is unarmed. These two modes together with modes `ALTmode` and `ATTmode` constitute the only mode class of the system, called `mcStatus`. Also, terms `tALTpresel`, `tCASpresel` and `tFPApresel` are introduced to characterize the states in which the desired altitude, calibrated air speed and flight path angle have been preselected, and the `CASmode` is also represented as a term. An additional term `tNear` is used to characterize the states in which the difference between the desired and actual altitudes is smaller than 1200.

For describing events, SCR provides a simple notation. The notation `@T(c) WHEN d` describes the event in which expression `c` becomes true, when `d` is `true` in the current state, i.e., it represents the expression $\texttt{c'} \wedge \texttt{c} \wedge \texttt{d}$, where the primed expression refers to the next state. If `d` is true, the ‘`WHEN`’ section is not written. Also, the event `CHANGED`(v) indicates that v has changed, i.e., it represents the expression $v' \neq v$. The table describing the mode transitions, as well as the values for terms `tALTpresel` and `tFPApresel` is the first one in Figure 1. This table corresponds to the merge of a mode transition table, describing the mode transitions, and two event tables. The values of the displays are defined by the next three small condition tables in Fig. 1. Finally, terms `tCASmode` and `tCASpresel` get defined by the bottom event table in Fig. 1. Term `tNear` has a definition which is independent of modes, conditions and events; its definition is simply $\texttt{mALTdesired} - \texttt{mALTactual} \leq 1200$.

Mode Class = mcStatus				
Old mode	Events	New mode	tALTpresel	tFPApresel
ATTmode	@T(mALTsw=on) WHEN (tALTpresel AND tNear)	ALTmode		false
	@T(mALTsw=on) WHEN (tALTpresel AND NOT tNear)	FPAarmed		
	@T(mFPAsw=on)	FPAunarmed	false	
	CHANGED(mALTdesired)		true	
	CHANGED(mFPAdesired)			true
ALTmode	@T(mATTsw=on)	ATTmode	false	false
	@T(mFPAsw=on)	FPAunarmed	false	
	CHANGED(mALTdesired)	ATTmode	false	false
	CHANGED(mFPAdesired)			true
	@T(mALTdesired = mALTactual)		false	
FPAarmed	@T(mATTsw=on) OR @T(mFPAsw=on)	ATTmode	false	false
	CHANGED(mALTdesired)	ATTmode	false	false
	CHANGED(mFPAdesired) AND NOT (mFPAdesired’ = mFPAactual’)			true
	@T(tNear) AND (mALTdesired’ = mALTdesired)	ALTmode		false
	@T(mFPAdesired = mFPAactual)			false
FPAunarmed	@T(mALTsw=on) WHEN (tALTpresel AND tNear)	ALTmode		false
	@T(mALTsw=on) WHEN (tALTpresel AND NOT tNear)	FPAarmed		
	@T(mATTsw=on) OR @T(mFPAsw=on)	ATTmode	false	false
	CHANGED(mALTdesired)		true	
	CHANGED(mFPAdesired) AND NOT (mFPAdesired’ = mFPAactual’)			true
	@T(mFPAdesired = mFPAactual)			false

	Conditions	
	tALTpresel	NOT tALTpresel
cALTdisplay =	mALTdesired	mALTactual

	Conditions	
	tCASpresel	NOT tCASpresel
cCASdisplay =	mCASdesired	mCASactual

	Conditions	
	tFPApresel	NOT tFPApresel
cFPAdisplay =	mFPAdesired	mFPAactual

Term = tCASmode			
	Events	tCASmode	tCASpresel
NOT tCASmode	@T(mCASsw=on)	true	
	CHANGED(mCASdesired)		true
tCASmode	@T(mCASsw=on)	false	false
	@T(mCASdesired = mCASactual)		false
	CHANGED(mCASdesired) AND NOT (mCASdesired’ = mCASactual’)		true

Fig. 1. Tabular specification of the autopilot system.

3 The Alloy and DynAlloy Modeling Languages

In the description of the autopilot system, the datatypes associated with the variables are obvious: numeric values range over integers, states for switches can be characterized by boolean values, and the possible values for the mode class `mcStatus` can be the defined modes for the system. In Alloy, these datatypes would be defined by *signatures*. For instance, the datatype associated with mode class `mcStatus` can be straightforwardly defined in Alloy in the following way:

```
abstract sig StatusMode { }
one sig ALTmode, ATTmode, FPAmode extends StatusMode { }
```

Abstract signatures have as their only associated elements, those of their non abstract “subsignatures”. The modifier `one` forces the corresponding signatures to have exactly one element, i.e., to be singletons. Thus, the above Alloy specification defines an enumerated set. More complex data domains can also be defined via signatures, using typed *fields*. For instance, we can define a signature to characterize the *state* associated with the autopilot system, composed of monitored and controlled variables, mode classes and state dependent terms, in the following way:

```
sig State {
 -- Monitored Variables
 mALTactual, mCASactual, mFPAactual, mALTdesired, mCASdesired,
 mFPAdesired: Int,
 mALTsw, mATTsw, mCASsw, mFPAsw: SwitchState,
 -- Controlled variables
 cALTdisplay, cCASdisplay, cFPAdisplay: Int,
 -- Mode classes
 mcStatus: StatusMode,
 -- Terms
 tARMED, tCASmode, tALTpresel, tCASpresel, tFPApresel: Boolean
}
```

Fields, which can have relational types, are interpreted as relations from the set associated with the signature in which the field is defined to the relation given as a type of the field. Thus, for instance, field `mcStatus` in signature `State` is interpreted as a relation from `State` to `StatusMode`.

Using signatures and fields, it is possible to build more complex expressions denoting relations, with the aid of the Alloy operators. Operator $\sim$ denotes relational transposition, $*$ denotes reflexive-transitive closure, and ^ denotes transitive closure of a binary relation. Operator $+$ denotes union, & denotes intersection, and dot (.) denotes composition of relations, generalized to n-ary relations and having relational image as a special case. In all cases, the typing must be adequate. Formulas are built from expressions. Binary predicate in checks for inclusion, while $=$ checks for equality. From these (atomic) formulas we define more complex formulas using standard first-order connectives and quantifiers. Negation is denoted by !. Conjunction, disjunction and implication are denoted by &&, || and =>, respectively. Finally, quantifications have the form some $a : A \mid \alpha(a)$ and all $a : A \mid \alpha(a)$. These formulas can be used in order to describe assumed as well as intended properties of the models. Parameterized formulas, which can be used for describing properties, can be written in Alloy using *predicates*. For instance, we can define a predicate for characterizing event `@T(mATTsw = on)`, as follows:

```
pred Ev_TmALTswOn(s,s': State) { s.mALTsw != on && s'.mALTsw = on }
```

Assumed properties of the specified data domains can be given as *facts* in Alloy. We can use facts for characterizing the values of terms or other variables defined via condition tables in a straightforward way. For our presented example, the values associated with controlled variables `cALTdisplay`, `cCASdisplay` and `cFPAdisplay` can be enforced using a fact, in the following way:

```
fact {
 all s: State | (s.tALTpresel = trueValue =>
                                       s.cALTdisplay = s.mALTdesired else
                                       s.cALTdisplay = s.mALT) &&
                (s.tCASpresel = trueValue =>
                       s.cCASdisplay = s.mCASdesired else
                                       s.cCASdisplay = s.mCASactual) &&
                (s.tFPApresel = trueValue =>
                       s.cFPAdisplay = s.mFPAdesired else
```

```
                                        s.cFPAdisplay = s.mFPAactual)
}
```

Intended properties, those to be checked, are defined in Alloy using *assertions*. For instance, we can consider the following assertion, corresponding to a disjointness check for the mode transition table associated with `mcStatus`:

```
assert DisjointnessCheck {
all s, s': State | ! (s.mcStatus = ATTmode &&
(Ev_TmALTswOn[s,s'] && s.tALTpresel = trueValue && s.tNear = trueValue) &&
(Ev_TmALTswOn[s,s'] && s.tALTpresel = trueValue && !(s.tNear = trueValue))
}
```

In Alloy, operations over the defined domains are specified using *predicates*. DynAlloy, on the other hand, incorporates the notion of *action* for specifying operations. Atomic actions, which are the basic units for specifying state change, are defined via pre- and post-conditions. This kind of description for an action indicates that, for the action to be executed, its precondition must be true, and in this case the state resulting from the execution of the action satisfies the postcondition. As a simple example, consider the following action, which characterizes the change of the monitored variable `mALTactual` (and the arbitrary change of the event-dependent terms and controlled variables):

```
act mALTactualChange[s: State] {
pre { }
post { s'.mALTactual != s.mALTactual && s'.mCASactual = s.mCASactual &&
       s'.mFPAactual = s.mFPAactual && s'.mALTsw = s.mALTsw &&
       s'.mATTsw = s.mATTsw && s'.mCASsw = s.mCASsw &&
       s'.mFPAsw = s.mFPAsw && s'.mALTdesired = s.mALTdesired &&
       s'.mCASdesired = s.mCASdesired && s'.mFPAdesired = s.FPAdesired }
}
```

Atomic actions can be composed to form *composite* actions (also called programs). These are built using sequential composition (`;`), nondeterministic choice (`+`), test (`[f]?`, an action that does not modify the state but can only be executed when `f` is true) and iteration (`*`). For example, if we consider atomic actions describing the change of each of the monitored variables (as we did above for monitored variable `mALTactual`), then the following composite action characterizes the change of *one* of the monitored variables:

```
program monitoredVarChange[s: State] {
 mALTactualChange[s] + mCASactualChange[s] + mFPAactualChange[s] +
 mALTswChange[s] + mATTswChange[s] + mCASswChange[s] + mFPAswChange[s] +
 mALTdesiredChange[s] + mCASdesiredChange[s] + mFPAdesiredChange[s]
}
```

This language for describing composite actions is what we are primarily interested in exploiting. DynAlloy also allows the specifier to write *assertions* associated with his programs, i.e., intended properties of the executions of the programs, to be checked. These properties to be checked are also given in the form of partial correctness assertions, i.e., by pre- and post-conditions. For example, the following DynAlloy assertion expresses that the above program cannot change `mALTactual` and `mCASactual` at the same time:

```
assertCorrectness[s:State] {
pre { }
program monitoredVarChange[s]
post { !(s'.mALTactual != s.mALTactual && s'.mCASactual != s.mCASactual)}
}
```

Alloy assertions can be automatically analyzed using the Alloy Analyzer. The mechanism for analysis is based on SAT solving. Basically, given a system specification and a statement about it, the Alloy tool exhaustively searches for a counterexample of this statement under the assumptions of the system description, by reducing the problem to the satisfiability of a propositional formula. Since the Alloy language is first-order, the search for counterexamples has to be performed up to a certain bound k in the number of elements in the universe of the interpretations. Thus, in order to check an assertion, the user has to provide bounds for the number of elements in the domains (associated with signatures) of the specification. Obviously, this analysis is not a decision procedure, since it cannot be used in general to guarantee the absence of counterexamples for a theory [10]. Nevertheless, it is useful in practice, since it allows one to discover counterexamples of intended properties, and if none is found, gain confidence about our specifications. This is similar in spirit to testing, since one checks the truth of a statement for a number of cases; however, as explained in [11], the scope of the technique is much greater than that of testing, since the space of cases examined (usually in the order of billions[4]) is beyond what is covered by testing techniques, and it does not require one to manually provide test cases.

DynAlloy assertions can also be analyzed automatically, by means of the same mechanism. Essentially, the DynAlloy Analyzer translates a DynAlloy assertion into an Alloy specification, which then can be analyzed using the Alloy Analyzer. In order to do so, the DynAlloy Analyzer needs, besides the bounds for the domains, an extra bound for iteration. This extra bound is used by the DynAlloy Analyzer to “unroll” the iterations in the program to be checked.

4 Characterizing Tables in DynAlloy

Part of the characterization of tables in DynAlloy has already been introduced in the previous section. First, type definitions, including the definition of signature `State`, are defined as shown in previous sections. The state of the system is composed of system variables, mode classes and terms. Second, each of the events mentioned in the tables gives rise to a corresponding predicate definition. Third, event independent terms and controlled variables, defined by condition tables, are constrained in Alloy using facts, as shown before for the displays. These facts are automatically synthesized from the condition tables. Fourth, mode transitions, described in a mode transition table, give rise to an Alloy predicate characterizing the transitions. In our case, this predicate is the following:

[4] This is so because, even for simple specifications and relatively small bounds, the number of bounded possible instances of the model, i.e., the cases to be examined, can be very large, easily reaching billions of possible instances [11].

```
pred NEXTmcStatus(s, s': State) {
s.mcStatus = ALTmode &&
                   (Ev_TmATTswOn[s,s'] || Ev_ChangedmALTdesired[s,s']) =>
                                                         s'.mcStatus = ATTmode else
(s.mcStatus = ALTmode && Ev_TmFPAswOn[s,s'] => s'.mcStatus = FPAmode else
(s.mcStatus = ATTmode && Ev_TmALTswOnWhentALTpreselAndtNear[s,s'] =>
                                                         s'.mcStatus = ALTmode else
(s.mcStatus = ATTmode && (Ev_TmFPAswOn[s,s'] ||
                          Ev_TmALTswOnWhentALTpreselAndNottNear[s,s']) =>
                                                         s'.mcStatus = FPAmode else
(s.mcStatus = FPAmode && (Ev_TmALTswOnWhentALTpreselAndtNear[s,s'] ||
                          Ev_TtNearWhentARMED[s,s'])
                                                      => s'.mcStatus = ALTmode else
(s.mcStatus = FPAmode && (Ev_TmATTswOn || Ev_TmFPAswOn ||
                          Ev_ChangedmALTdesiredWhentARMED[s,s'])
                                                           => s'.mcStatus = ATTmode
                                                 else s'.mcStatus = s.mcStatus)))))
}
```

Notice that we are making use of the predicates associated with the events. Clearly, predicate `NEXTmcStatus` corresponds to the formula specified using a tabular notation in SCR. We follow a similar process for controlled variables and terms defined by event tables. For instance, we will have a predicate associated with the event table defining `tCASmode`, etc. Finally, using these predicates we define a single DynAlloy action, called `stateChange`, as follows:

```
act stateChange[s: State] {
pre {  }
post { monitoredVariableChange[s,s'] && NEXTmcStatus[s,s'] &&
   NEXTtARMED[s,s'] && NEXTtCASmode[s,s'] &&
   NEXTtALTpresel[s,s'] && NEXTtCASpresel[s,s'] && NEXTtFPApresel[s,s'] }
}
```

where `monitoredVariableChange` is an Alloy predicate characterizing the change of a monitored variable. Other elements of a tabular specification, such as initial states, are also straightforwardly characterized in DynAlloy. The important point here is that the generation of the DynAlloy specification corresponding to the tabular descriptions is fully automated.

With the DynAlloy characterization of the tabular specification, we can do various analyses. For instance, we can use the Alloy Analyzer for checking disjointness and completeness associated with tables (an example of this is assertion `DisjointnessCheck` given in the previous section). We can also check properties of *all* executions, using the DynAlloy Analyzer. For instance, we could check that, whenever the system is in the `ALTmode`, the altitude display shows the desired altitude. This is specified using the following DynAlloy assertion:

```
assert basicProgram {
  assertCorrectness[s: State] {
  pre = { initialState[s] }
  program = { stateChange[s]* }
  post = { (s'.mcStatus = ALTmode) => (s'.cALTdisplay = s'.mALTdesired) }
  }
}
```

where `initialState` is a predicate describing the initial state for the system (also synthesized from the tabular specification).

5 Specifying and Analyzing Sets of Executions via DynAlloy Programs

Most of the analysis techniques associated with tabular specifications, including the SAT based analysis described in the previous section, have their efficiency tied to the size of the specification (or, put in a different way, to the number of possible runs associated with the specification). We propose here a notation for describing subsets of the executions of a tabular specification. The notation is essentially that of DynAlloy programs, complemented with a way of restricting action `stateChange`, our only atomic action, representing a change in the state of the system as defined by the tables. Intuitively, the *conditioned* atomic action definition:

```
stateChange<<f[s,s']>>[s]
```

where `f[s,s']` is a formula referring to the pre and post states, corresponds to action `stateChange` occurring, with `f` also taking place in the transition. For example, the following actions:

```
stateChange<<@T(mALTsw = on)>>[s]  stateChange<<NOT @T(mALTsw = on)>>[s]
```

correspond to action `stateChange`, restricted to the facts that the `mALTsw` switch must be switched on in the change, and must *not* be switched on in the change, respectively. If an action `a` is defined by pre and postconditions `pre_a` and `post_a`, then its semantics is the relation associated with the formula $\mathtt{pre\_a}[s] \wedge \mathtt{post\_a}[s, s']$. Action `a<<f[s,s']>>` has instead as its semantics the relation associated with the formula $\mathtt{pre\_a}[s] \wedge \mathtt{post\_a}[s, s'] \wedge \mathtt{f}[s, s']$. Since the restrictions for atomic action `stateChange` are provided by the user, the SCR notation is employed for expressing them. In this way, the specifier used to the tabular notation will not need to deal directly with (Dyn)Alloy. The idea is, of course, that (Dyn)Alloy should be used as a backend for analysis.

Let us consider as a first example of the use of the notation the following. If none of the switches in the panel are switched, then the mode remains being that of the initial state, namely `ATTmode`. This is expressed as follows:

```
assert SimpleCheck {
 assertCorrectness[s: State] {
  pre = { initialState[s] }
  program = { stateChange<<!@T(mATTsw = switchOn)
           AND !@T(mFPAsw = switchOn) AND !@T(mALTsw = switchOn)>>[s]* }
  post = { s'.mcStatus = ATTmode }
 }
}
```

Another example is the following. Suppose that one wants to check if, once the ALTmode is on, if no switch is switched afterwards then either the current mode is ALTmode, or is back to mode ATTmode (i.e., the ALTmode is deactivated because the desired altitude was reached). This is expressed as follows:

```
assert ALTmodeDisengagedWhenALTreached {
  assertCorrectness[s: State] {
  pre = { initialState[s] }
  program = { stateChange[s]* ; [ s.mcStatus = ALTmode]? ;
            stateChange<<!@T(mATTsw = switchOn) AND !@T(mFPAsw = switchOn)
                          AND !@T(mALTsw = switchOn)>>[s]* }
  post = { s'.mcStatus = ALTmode || s'.mcStatus = ATTmode }
  }
}
```

These examples use conditioned actions, test actions, iteration and sequential composition. Let us now consider the following assertion: If the mALTsw is pressed below 1200 from the desired altitude, then if the panel is not touched and the airplane reaches an altitude higher than or equal to the desired altitude, the airplane moves to ALTmode and the altitude display shows the current altitude. This is expressed in the following way:

```
assert BackToALTWhenALTpassed {
  assertCorrectness[s: State] {
  pre = { initialState[s] }
  program = { stateChange[s]* ;
              [ s.mALTdesired - s.mALTactual > 1200 ]? ;
              stateChange<<@T(mALTsw = switchOn)>>[s] ;
           stateChange<<!@T(mATTsw = switchOn) AND !@T(mFPAsw = switchOn)
                          AND !@T(mALTsw = switchOn) AND
                          NOT Changed(mALTdesired) AND
                          NOT Changed(mFPAdesired) AND
                          NOT Changed(mCASdesired)>>[s]* ;
              [ s.mALTactual > s.mALTdesired]? ; }
  post = { s'.mcStatus = ALTmode && s'.cALTdisplay = s'.mALTactual }
  }
}
```

Via programs, we *externalize* the specification of control flow from tabular specifications. Notice that the number of possible executions of these programs if iterated just a few times is huge, due to the various different branches these can follow in different iterations; this is beyond the scope of what can practically be done by manually defining executions for simulation, for instance by using the SCR toolset's simulator. Notice that tables in a specification describe a labeled transition relation, and a system's behaviour is understood as the reflexive-transitive closure of this relation. For these relations to adequately describe the wanted behaviours associated with any of the properties to be checked given in this section, it would be necessary to include control variables (or new modes) whose values rule out undesired control flows. Some of these variables can be deemed unnecessary by using an action language allowing us to externally define complex behaviours, as the one we propose.

Although in this paper we analyze *programs* over tabular specifications using SAT-solving, via the (Dyn)Alloy and DynAlloy analyzers, the notation is not limited to this usage. For instance, it is relatively straightforward to translate assertions of the kind shown in this section to the input languages of other analysis tools, in particular model checking tools.

6 Synthesis for Conditioned Atomic Actions

According to the semantics of conditioned atomic actions, it is clear that we can straightforwardly synthesize new DynAlloy atomic actions for each of the conditioned atomic actions used by the specifier. For instance, conditioned atomic action `stateChange<<@T(mcStatus = ATTmode)>>[s,s']` would lead to the following DynAlloy atomic action:

```
act stateChangeCondEv_TmcStatusATTmode[s: State] {
pre {  } post { postStateChange[s,s'] && Trans(@T(mcStatus = ATTmode)) }
}
```

where `postStateChange` is the original postcondition of action `stateChange` and `Trans(@T(mcStatus = ATTmode))` corresponds to the mapping of `@T(mcStatus = ATTmode)` into Alloy's syntax (and the predicates associated with events introduced). However, this has some disadvantages. Conditioned actions, which should be restrictions of the original atomic actions, are actually more complex. This has a negative impact with respect to analysis, as it will be made clearer in the next section. Therefore, we consider an alternative mechanism for generating the corresponding DynAlloy definition of a conditioned atomic action. This mechanism makes use of the information associated with the wellformedness of tabular specifications (in particular, disjointness), and the semantics of tables.

Assuming that we have already checked the wellformedness of our tabular specifications, as defined in [6], the process for synthesizing a DynAlloy atomic action from a conditioned action `stateChange[e]` works by identifying a subset of the events used in the tables, the set of *incompatible* events with respect to `e`. We restrict the construction to atomic events of the form `@T(v = c)`, `@F(v = c)`, `CHANGED(x)`, `@T(v = c) WHEN cond`, `@F(v = c) WHEN cond`, and combinations of these using conjunction, disjunction and negation. Furthermore, in order to make the process efficient, it is based on the sole syntactic analysis of the events used in the tables, i.e., without resorting to the use of SAT solving to check whether two events are incompatible or not. We will describe the rules for constructing the set of incompatible events for `@T(x = c)`, but the reader can straightforwardly generalize the principles used in this construction to the other events. Let `e` be `@T(alpha)`. The events incompatible with `e` are the following: *(i)* `@F(alpha)` is incompatible with `e`. *(ii)* If `alpha` is of the form `x = c`, with `x` a variable and `c` a constant, then `@T(x = c')` and `@T(x = c') WHEN cond` are incompatible with `e`, for every constant `c'` different from `c`. *(iii)* If `alpha` is of the form `x = c`, with `x` a variable, term or mode class, then then `@T(y = c')`, `@T(y = c') WHEN cond`, `@F(y = c')`, `@F(y = c') WHEN cond` and `CHANGED(y)` are incompatible with `e`, for every variable, term of mode class `y` which is not a predecessor nor a successor of x in the symbol dependency graph, and for every expression c'. *(iv)* Conjunctions involving any of the above cases are also incompatible with e. *(v)* Disjunctions whose all composing disjuncts correspond to the above cases are incompatible with e.

These incompatible events are guaranteed not to occur simultaneously with e. The reason for this in the first and second of the above cases is obvious. The third is based on the observation that, if x changed its value, then all the symbols which do not depend on x and of which x does not depend, cannot have changed. This has to do with the assumption that events start with the change of a single monitored variable, and the propagation of changes to the symbols depending on this variable. Notice also that the third of the above rules has as a special case that, if a monitored variable changes in a transition step, then none of the other monitored variables changes in the same transition (recall that we assume that the tabular specification is valid, implying that the symbol dependency graph is acyclic). The reason for the last two rules are obvious due to the semantics of conjunction and disjunction, respectively.

Once the set of incompatible events is constructed for e, the process is straightforward. First, we remove the cases in the tabular specifications involving events incompatible with e. Second, in the tables where disjointness conditions apply, the alternatives to e are removed. We then generate the “stateChange” formula corresponding to the resulting tables. These formulas can be much simpler than the original **stateChange** postcondition. For instance, the combined mode transition/event table given previously, restricted according to stateChange(@T(mATTsw=on)) is reduced to the following:

Mode Class = mcStatus				
Old mode	Events	New mode	tALTpresel	tFPApresel
ALTmode	@T(mATTsw=on)	ATTmode	false	false
FPAarmed	@T(mATTsw=on) OR @T(mFPAsw=on)	ATTmode	false	false
FPAunarmed	@T(mATTsw=on) OR @T(mFPAsw=on)	ATTmode	false	false

The resulting tables might not satisfy some of the wellformedness conditions (e.g., when we remove a row, we might lose completeness); however, this is not a concern, since the resulting tables are guaranteed to be equivalent to the original, if restricted to the occurrence of event e, used to “restrict” the tables.

7 Analyzing Programs over Tabular Specifications

In order to assess our approach, we have conducted some case studies, including two different versions of the autopilot system, taken from [1, 2], and a tabular specification of a mini FM radio. In order for the Alloy Analyzer to be able to handle integer values, we had to consider abstractions of the altitude, etc, using smaller integers (smaller than 16). Although this is a limitation particularly associated with the Alloy Analyzer, this kind of abstraction process is also typical of other automated analysis techniques, such as model checking. The results of the experiments, which were carried out using an Intel Core 2 Duo of 2.2Ghz with 2GB of RAM, running the Alloy Analyzer 4.1.8 over Mac OS X, were positive. In the cases of the somewhat more preliminary specification of the autopilot system taken from [1] and the specification of the mini FM radio, the Alloy

Analyzer allowed us to find various errors in the specifications, including consistency errors, errors related to misinterpretations of events, and transcription mistakes. For the better developed specification of the autopilot system in [2], we carried out experiments involving the execution programs given above. We compared the straightforward (SG) and based on event compatibility (ECG) approaches to generating definitions for conditioned atomic actions, with the latter showing a better performance compared to the former, as expected. We summarize the analysis times for two of the assertions, namely `SimpleCheck` and `BackToALTWhenALTpassed`, in the table below (times are in seconds). We found out using the (Dyn)Alloy Analyzer that this last property is, contrary to what we expected, invalid. The first counterexample obtained had to do with the altitude changing arbitrarily, and was solved by requiring it to be increased/decreased in units (which must divide the representation of 1200, used in `tNear`). The second counterexample obtained exhibited the following situation: if the airplane gets into `FPAunarmed` mode after the desired altitude has been altered, the system will stop considering that the altitude has been preselected; thus, when `mALTsw` is pressed, the event is ignored and the system will continue to be in `FPAunarmed` mode. Obviously, this has to do with a misinterpretation of the relationship between `tNear` and `tALTpresel`, which we can correct in the program by requiring in the intermediate test action not only that `tNear` be false, but also that `tALTpresel` is true. `SimpleCheck` and the corrected `BackToALTWhenALTpassed` (also included in the table) properties are valid within the provided bounds, requiring exhaustive explorations of cases for the corresponding bounds. Also, `BackToALTWhenALTpassed` contains two loops, leading to longer runs for the corresponding loop unrolls and longer analysis times, as it can be observed in the table.

Loop unrolls	`SimpleCheck`		`BackToALTWhenALTpassed`		`BackToALTWhenALTpassed` corrected	
	SG	ECG	SG	ECG	SG	ECG
5	.563	.602	2.740	1.233	17.442	16.181
10	1.855	1.255	6.253	6.305	106.018	122.003
20	6.184	4.510	84.421	33.518	$> 60'$	$> 60'$
40	46.749	21.648	576.402	170.946	$> 60'$	$> 60'$
80	422.722	285.784	$> 60'$	$> 60'$	$> 60'$	$> 60'$

8 Conclusions

We have proposed a formal notation for describing behaviours over tabular specifications of requirements. This formalism enabled us to describe classes of *computations*, in the sense of particular combinations of the atomic transitions specified by tables. As opposed to previous approaches, our approach allows for the description of a wider class of combinations of the transitions defined by tables, by means of a rich operational language. This language is based on DynAlloy, an extension of the Alloy formal specification language that incorporates actions, both atomic and composite, in order to specify state change in a suitable way. Our approach comes equipped with some tool support, since, as we showed in the paper, the DynAlloy Analyzer [4] can be used to validate partial correctness assertions over tabular specifications via a SAT based mechanism (by indirectly employing the Alloy Analyzer).

The proposed notation for characterizing particular combinations of the transitions specified by tables is not restricted to SAT-based analysis. The powerful tool support associated with existing tabular notations for requirements [7, 3, 8, 13] might benefit from the presented approach. In particular, starting from the DynAlloy assertions (accompanied by programs) of the kind presented in the paper, one can generate corresponding specifications in the input languages of model checkers. Thus, one could apply model checking, in the way that the SCR toolset applies it, to analyze the behaviours corresponding to these programs. We are confident that this would contribute to the efficiency of the analysis and the convenience of the user, by allowing the specifier to concentrate verification on particular sets of runs (specified by programs over tables) in a declarative manner. It is our aim and part of our work in progress to apply some of these analysis techniques with a focus on the runs specified by programs in our notation, rather than on the global set of behaviours.

References

1. R. Bharadwaj and C. Heitmeyer, *Applying the SCR Requirements Specification Method to Practical Systems: A Case Study*, 21st Software Engineering Workshop, NASA GSFC, 1996.
2. R. Bharadwaj and C. Heitmeyer, *Applying the SCR Requirements Method to a Simple Autopilot*, in Proc. of the Fourth NASA Langley Formal Methods Workshop, 1997.
3. T. Bultan and C. Heitmeyer, *Analyzing Tabular Requirements Specifications using Infinite State Model Checking*, in Proc. of MEMOCODE 2006, 2006.
4. M. Frias, J. P. Galeotti, C. López Pombo and N. Aguirre, *DynAlloy: Upgrading Alloy with Actions*, in Proc. of ICSE '05, ACM Press, 2005.
5. A. Gargantini and C. Heitmeyer, *Using Model Checking to Generate Tests from Requirements Specifications*, in Proc. of ESEC/FSE '99, LNCS, Springer, 1999.
6. C. Heitmeyer, A. Bull, C. Gasarch and B. Labaw, *SCR*: A Toolset for Specifying and Analyzing Requirements*, in Proc. of COMPASS '95, Gaithersburg, 1995.
7. C. Heitmeyer, R. Jeffords and B. Labaw, *Automated consistency checking of requirements specifications*, ACM Trans. on Soft. Eng. and Methodology, 5(3), 1996.
8. C. Heitmeyer, M. Archer, R. Bharadwaj and R. Jeffords, *Tools for constructing requirements specifications: the SCR Toolset at the age of nine*, Computer Systems: Science & Engineering, 20(1), 2005.
9. K. Heninger, J. Kallander, D. Parnas and J. Shore, *Software Requirements for the A-7E Aircraft*, NLR Memorandum Report 3876, US Naval Research Lab., 1978.
10. D. Jackson, *Alloy: a lightweight object modelling notation*, ACM Trans. on Soft. Eng. and Methodology, 11(2), 2002.
11. D. Jackson, *Software Abstractions: Logic, Language, and Analysis*, MIT Press, 2006.
12. N. Leveson, M. Heimdahl, H. Hildreth and J. Reese, *Requirements Specifications for Process-Control Systems*, IEEE Trans. on Software Engineering, 20(9), 1994.
13. S. Owre, J. Rushby and N. Shankar, *Analyzing Tabular and State-Transition Specifications in PVS*, in Proc. of TACAS '97, LNCS, Springer, 1997.