Proving feature non-interaction with Alternating-Time Temporal Logic

Franck Cassez[1], Mark Dermot Ryan[2], and Pierre-Yves Schobbens[3]

[1] IRCCyN – BP 92101, 1 rue de la Noë, 44321 Nantes cedex 03, France. Franck.Cassez@ircyn.ec-nantes.fr
[2] School of Computer Science, University of Birmingham, Edgbaston, Birmingham B15 2TT, England. mdr@cs.bham.ac.uk
[3] Institut d'Informatique, Facultés Universitaires de Namur, Rue Grandgagnage 21, 5000 Namur, Belgium. pys@info.fundp.ac.be

1 Introduction

Feature Interaction. When engineers design a system with features, they wish to have methods to prove that the features do not interact in ways which are undesirable. A considerable literature is devoted to this 'feature interaction problem' [13,5]. One approach to demonstrating that features do not interact undesirably is to equip them with properties which are intended to hold of a system having the feature [17]. In this view, a feature is a pair (F, ϕ) consisting of the implementation of the feature F and a set of properties ϕ. Integrating a feature (F, ϕ) with a base system S consists of modifying the base system in the way described by the feature implementation and obtaining $S + F$. The integration is deemed successful if the resulting system satisfies the set of properties ϕ corresponding to the feature. Evidence that a feature (F_1, ϕ_1) does not negatively interact with feature (F_2, ϕ_2) may be obtained by verifying that introducing F_2 in $S + F_1$, (obtaining $S + F_1 + F_2$) does not destroy the properties ϕ_1 previous introduced by feature F_1, and vice versa.

Model-Checking. Model checking [10] may be used to show that the featured system has the desired properties (cf. [6,17,16,14]). However, the approach described may involve checking the same property again and again each time a new feature is introduced, to check that a previously introduced feature has not been broken. Since model checking is computationally expensive, it is worthwhile to find methods which avoid these re-checks. For example, we may be able to prove generally that a certain feature does not destroy properties in a certain class. This would obviate the need to re-check properties in that class when the feature is introduced.

Users' Viewpoints and Conservative Features. A general result defining a class of properties which are provably not broken by the introduction of a new feature could be inspired by a number of intuitions. In this paper, we develop such a result based on the idea that a feature which *adds* to the

capabilities of a *user* (and does not subtract from them) should not break properties which *assert* capabilities of this user. Let us look at the notion of capabilities in more detail.

A telephone system is usually made of a number of users and a network managing the calls. Many features of telephone systems are designed so that they add to the capabilities (or *powers*), of the subscribing user, without subtracting from them. For example, a user j who subscribes to call-forwarding now has the power to forward his or her calls to another user, but has not lost any capabilities in the process. When this is the case, we say that the feature is j-conservative. More generally, if U is a set of users, a feature is U-conservative if any behaviour of the system which U could enforce before the feature was added, U can also enforce it after the feature is added. The principal idea in this paper is that a U-conservative feature does not break properties which assert capabilities of the group U of users.

Framework. To formalise this intuition, we propose to use Alternating-time Temporal Logic [4] (ATL), which allows us to describe precisely the properties of the different *agents* (or users) involved in a system, and the *strategies* they have for achieving their goals. ATL is a branching temporal logic based on game theory. It contains the usual temporal operators (next, always, until) plus cooperation modalities $\langle\langle A\rangle\rangle\phi$, where A is a set of players. This modality quantifies over the set of behaviours and means that A has a collective strategy to enforce ϕ, whatever the choices of the other players. ATL generalises CTL, and similarly ATL* generalises CTL*, μ-ATL generalises the μ-calculus. These logics can be model-checked by generalising the techniques of CTL, often with the same complexity.

Outline of the paper. In section 2 we recall the basic concepts of ATL and ATL* and their semantics on ATSs. This section also introduces the Mocha-like language we use to describe reactive systems. The next section 3 which is the core of the paper describes the feature construct for our Mocha-like language and states the properties-preserving theorem. Finally in section 4 we discuss some directions for future work.

2 Alternating-time temporal logic

Alternating-time temporal logic (ATL) is based on CTL. Let us first recall a few facts about CTL. CTL [9] is a branching-time temporal logic in which we can express properties of reactive systems. For example, properties of cache-coherence protocols [15], telephone systems [17], and communication protocols have been expressed in CTL. One problem with CTL is that it does not distinguish between different sources of non-determinism. In a telephone system, for example, the different sources include individual users, the environment, and internal non-determinism in the telephone exchange. CTL

provides the A quantifier to talk about all paths, and the E quantifier to assert the existence of a path. Aψ means that, no matter how the non-determinism is resolved, ψ will be true of the resulting path. Eψ asserts that, for at least one way of resolving the non-determinism, ψ will hold. But because CTL does not distinguish between different types of non-determinism, the A quantifier is often too strong, and the E quantifier too weak. For example, if we want to say that *user i can converse with user j*, CTL allows us to write the formulas

$$\texttt{A}\Diamond\texttt{talking}(i,j), \quad \texttt{E}\Diamond\texttt{talking}(i,j).$$

The first one says that in all paths, somewhere along the path there is a state in which i is talking to j, and is clearly much stronger than the intention. The second formula says that there is a path along which i is eventually talking j. This formula is weaker than the intention, because to obtain that path we may have to make choices on behalf of all the components of the system that behave non-deterministically. What we wanted to say is that users i and j can resolve their non-deterministic choices in such a way that, no matter how the other users or the system or the environment behaves, all the resulting paths will eventually have a state in which i is talking j. Of course, the fact that i is talking to j requires the cooperation of j. This subtle differences in expressing the properties we want to check can be captured accurately with ATL.

Alternating-time temporal logic (ATL) [4] generalises CTL by introducing *agents*, which represent different sources of non-determinism. In ATL the A and E path quantifiers are replaced by a unique path quantifier $\langle\langle A\rangle\rangle$, indexed by a subset A of the set of agents. The formula $\langle\langle A\rangle\rangle\psi$ means that the agents in A can resolve their non-deterministic choices such that, no matter how the other agents resolve their choices, the resulting paths satisfy ψ. We can express the property that user i has the power, or capability, of talking to j by the ATL formula[1]

$$\langle\langle i\rangle\rangle\Diamond\texttt{talking}(i,j).$$

We read $\langle\langle A\rangle\rangle\psi$ as saying that the agents in A can, by cooperating together, force the system to execute a path satisfying ψ. If A is the empty set of agents, $\langle\langle A\rangle\rangle\psi$ says that the system will execute ψ without the cooperation of any agents at all; in other words, $\langle\langle\emptyset\rangle\rangle\psi$ is equivalent to Aψ in CTL. Dually, $\langle\langle\Sigma\rangle\rangle\psi$ (where Σ is the entire set of agents) is a weak assertion, saying that if all the agents conspire together they may enforce ψ, which is equivalent to Eψ in CTL.

2.1 ATL and ATL*

Let P be a set of atomic propositions and Σ a set of agents. The syntax of ATL is given by

$$\phi ::= p \mid \top \mid \neg\phi_1 \mid \phi_1 \vee \phi_2 \mid \langle\langle A\rangle\rangle[\phi_1 \ \mathrm{U} \ \phi_2] \mid \langle\langle A\rangle\rangle\Box\phi_1 \mid \langle\langle A\rangle\rangle\bigcirc\phi_1$$

[1] We write $\langle\langle i\rangle\rangle\psi$ instead of $\langle\langle\{i\}\rangle\rangle\psi$.

where $p \in P$ and $A \subseteq \Sigma$. We use the usual abbreviations for $\rightarrow$, $\wedge$ in terms of $\neg$, $\vee$. The operator $\langle\langle\ \rangle\rangle$ is a path quantifier, and $\bigcirc$ (*next*), $\Box$ (*always*) and U (*until*) are temporal operators. The logic ATL is similar to the branching-time logic CTL, except that path quantifiers are parameterised by sets of agents. As in CTL, we write $\langle\langle A\rangle\rangle\Diamond\phi$ for $\langle\langle A\rangle\rangle[\top\ \mathrm{U}\ \phi]$.

While the formula $\langle\langle A\rangle\rangle\psi$ means that the agents in A can cooperate to make ψ true (they can "enforce" ψ), the dual formula $[[A]]\psi$ means that the agents in A cannot cooperate to make ψ false (they cannot "avoid" ψ). The formulas $[[A]]\Diamond\phi$, $[[A]]\Box\phi$, and $[[A]]\bigcirc\phi$ stand for $\neg\langle\langle A\rangle\rangle\Box\neg\phi$, $\neg\langle\langle A\rangle\rangle\Diamond\neg\phi$, and $\neg\langle\langle A\rangle\rangle\bigcirc\neg\phi$.

The logic ATL* generalises ATL in the same way that CTL* generalises CTL, namely by allowing path quantifiers and temporal operators to be nested arbitrarily.

For a subset $A \subseteq \Sigma$ of agents, the fragment $\langle\langle A\rangle\rangle$-ATL of ATL consists of ATL formulas whose only modality is $\langle\langle A\rangle\rangle$, and that does not occur within the scope of a negation. The $\langle\langle A\rangle\rangle$-ATL* fragment of ATL* is defined similarly.

2.2 Alternating transitions systems

Whereas the semantics of CTL is given in terms of transition systems, the semantics of ATL is given in terms of *alternating transition systems* (ATSs). An ATS over a set of atomic propositions P and a set of agents Σ is a triple $S = (Q, \pi, \delta, I)$, where Q is a set of states and $\pi : Q \rightarrow 2^P$ maps each state to the set of propositions that are true in it, and

$$\delta : Q \times \Sigma \rightarrow 2^{2^Q}$$

is a transition function which maps a state and an agent to a non-empty set of *choices*, where each choice is a non-empty set of possible next states. If the system is in a state q, each agent a chooses a set $Q_a \in \delta(q, a)$; the system will move to a state which is in $\bigcap_{a\in\Sigma} Q_a$. We require that the system is non-blocking and that the agents together choose a unique next state; that is, for every q and every tuple $(Q_a)_{a\in\Sigma}$ of choices $Q_a \in \delta(q, a)$, we require that $\bigcap_{a\in\Sigma} Q_a$ is a singleton. Similarly, the initial state is specified by $I : \Sigma \rightarrow 2^{2^Q}$. I maps each agent to a set of choices. The agents together choose a single initial state: for each tuple $(Q_a)_{a\in\Sigma}$ of choices $Q_a \in I(a)$, we require that $\bigcap_{a\in\Sigma} Q_a$ is a singleton.

For two states q and q', we say that q' is a *successor* of q if, for each $a \in \Sigma$, there exists $Q' \in \delta(q, a)$ such that $q' \in Q'$. We write $\delta(q)$ for the set of successors of q; thus,

$$\delta(q) = \bigcap_{a\in\Sigma} \bigcup_{Q\in\delta(q,a)} Q$$

A computation of S is an infinite sequence $\lambda = q_0, q_1, q_2 \ldots$ of states such that (for each i) q_{i+1} is a successor of q_i. We write $\lambda[0, i]$ for the finite prefix $q_0, q_1, q_2, \ldots, q_i$.

Often, we are interested in the cooperation of a subset $A \subseteq \Sigma$ of agents. Given A, we define $\delta(q, A) = \{\bigcap_{a \in A} Q_a \mid Q_a \in \delta(q, a)\}$. Intuitively, when the system is in state q, the agents in A can choose a set $T \in \delta(q, A)$ such that, no matter what the other agents do, the next state of the system is in T. Note that $\delta(q, \{a\})$ is just $\delta(q, a)$, and $\delta(q, \Sigma)$ is the set of singleton successors of q.

Example 1 ([4]). Consider a system with two agents "user" u and "telephone exchange" e. The user may lift the handset, represented as assigning value true to the boolean variable "offhook". The exchange may then send a tone, represented by assigning value true to the boolean variable "tone". Initially, both variables are false. Clearly, obtaining a tone requires collaboration of both agents.

We model this as an ATS $S = (Q, \pi, \delta, I)$ over the agents $\Sigma = \{u, e\}$ and propositions $P = \{\text{offhook,tone}\}$. Let $Q = \{00, 01, 10, 11\}$. 00 is the state in which both are false, 01 the state in which "offhook" is false and "tone" is true, etc. (thus, $\pi(00) = \emptyset$, $\pi(01) = \{tone\}$, etc.). The transition function δ and initial states I are as indicated in the figure.

$\delta(q,a)$	u	e
00	$\{\{00,01\},\{10,11\}\}$	$\{\{00,10\}\}$
10	$\{\{10,11\}\}$	$\{\{00,10\},\{01,11\}\}$
01	$\{\{00,01\},\{10,11\}\}$	$\{\{01,11\}\}$
11	$\{\{10,11\}\}$	$\{\{01,11\}\}$
I	$\{\{00,01\}\}$	$\{\{00,10\}\}$

Fig. 1. The transition function of the ATS.

2.3 Semantics

The semantics of ATL uses the notion of *strategy*. A *strategy* for an agent $a \in \Sigma$ is a mapping $f_a : Q^+ \to 2^Q$ such that $f_a(\lambda \cdot q) \in \delta(q, a)$ with $\lambda \in Q^*$. In other words, the strategy is a recipe for a to make its choices. Given a state q, a set A of agents, and a family $F_A = \{f_a \mid a \in A\}$ of strategies, the *outcomes* of F_A from q are the set $out(q, F_A)$ of all computations from q where agents in A follow their strategies, that is,

$$out(q_0, F_A) = \{\lambda = q_0, q_1, q_2, \cdots \mid \forall i,\ q_{i+1} \in \delta(q_i) \cap \big(\bigcap_{a \in A} f_a(\lambda[0, i])\big)\}.$$

If $A = \emptyset$, then $out(q, F_A)$ is the set of all computations, while if $A = \Sigma$ then it consists of precisely one computation.

The semantics of ATL* is as CTL*, with the addition of:

- $q \models \langle\langle A \rangle\rangle \psi$ if there exists a set F_A of strategies, one for each agent in A, such that for all computations $\lambda \in out(q, F_A)$ we have $\lambda \models \psi$.

Remark 1. To help understand the ideas of ATL, we state below some validities, and more surprising non-validities.

1. If $A \subseteq B$, then $\langle\langle A \rangle\rangle \psi \rightarrow \langle\langle B \rangle\rangle \psi$, and $[[B]]\psi \rightarrow [[A]]\psi$. Intuitively, anything that A can enforce can also be enforced by a superset B; and if anything that B is powerless to prevent cannot be prevented by a subset of B.
2. In CTL, A distributes over $\wedge$. But in general, $\langle\langle A \rangle\rangle(\psi_1 \wedge \psi_2)$ only implies $(\langle\langle A \rangle\rangle \psi_1) \wedge (\langle\langle A \rangle\rangle \psi_2)$. The first formula asserts that A can enforce $\psi_1 \wedge \psi_2$, while the second is weaker, asserting that A has a way to enforce ψ_1 and another, possibly incompatible, way to enforce ψ_2. Similarly, $\langle\langle A \rangle\rangle(\psi_1 \vee \psi_2)$ and $\langle\langle A \rangle\rangle \psi_1 \vee \langle\langle A \rangle\rangle \psi_2$ are different (for $A \neq \Sigma$). The first one asserts that A can enforce $\psi_1 \vee \psi_2$, but which of the two is true might be chosen by others. This is weaker than the second formula, which asserts that A can guarantee ψ_1, or A can guarantee ψ_2, but nobody can choose which. The strongest variant where A can choose, is expressed as: $\langle\langle A \rangle\rangle(\psi_1 \wedge \neg\psi_2) \wedge \langle\langle A \rangle\rangle(\psi_2 \wedge \neg\psi_1)$.
3. By repeating a cooperation inside a temporal operator, we weaken the formula, for instance: $\langle\langle A \rangle\rangle \Box\Diamond\phi \rightarrow \langle\langle A \rangle\rangle \Box \langle\langle A \rangle\rangle \Diamond\phi$. This is because the strategies F_A that A use in the outer modality may be adapted for the inner modality, by shifting its time: each $f'_a(x)$ is simply $f_a(\lambda \cdot x)$, where λ is the path linking the points of evaluation of the two modalities. (Note the CTL* validities $\mathrm{E}\Box\Diamond\phi \rightarrow \mathrm{E}\Box\mathrm{E}\Diamond\phi$ and $\mathrm{A}\Box\Diamond\phi \leftrightarrow \mathrm{A}\Box\mathrm{A}\Diamond\phi$.)

2.4 Guarded command language

ATSs may be described using a Mocha-like guarded command language. (Mocha [1] is the system modelling language used for ATL.) We illustrate this with the system S of the preceding section.

```
agent USER
  controls offhook;
  init
     offhook := false;
  update
     true -> ;
     true -> offhook := true;
endagent;
```

```
agent EXCH
  controls tone;
  init
     tone := false ;
  update
     true -> ;
     offhook -> tone := true;
endagent;
```

The `init` clause gives the initial values of variables (if they are not mentioned, their initial values are selected non-deterministically). The `update` clause consists of a set of guarded commands, consisting of a guard (before the arrow) and a command (after the arrow). The agents are run in parallel. At each step, the guards in the agent are evaluated, and the agent chooses one which evaluates to true. The command corresponding to that guard is executed. If a variable is not assigned to in a command, it preserves its old value. In particular, if the command is empty, nothing changes: the cryptic-looking command `true ->` simply allows the user to wait. Every variable is controlled by precisely one agent; only the controlling agent can assign to the variable. Agents may refer to variables which are controlled by other agents (for example, `EXCH` refers to `offhook` which is controlled by `USER`).

2.5 Simulation and trace containment

It is known in CTL that if a transition system S' *simulates* another one S, written $S \leq S'$, then all ACTL* formulas which hold of S also hold of S'. (ACTL* is the universal fragment of CTL*, i.e. the fragment in which the only path quantifier is A, and no negations are allowed which include A in their scope.)

A similar result holds for ATL* [3]. Instead of a single notion of simulation, they define a notion indexed by a set of agents A. Let $S = (Q, \pi, \delta, I)$ and $S' = (Q', \pi', \delta', I')$ be ATSs over agents Σ, with $P \subseteq P'$. For a subset $A \subseteq \Sigma$ of agents, a relation $H \subseteq Q \times Q'$ is an *A-simulation* from S to S' if[2]:

- For every set $T \in I(A)$, there exists a set $T' \in I'(A)$ such that for every set $R' \in I'(\Sigma - A)$ there exists a set $R \in I(\Sigma - A)$ such that $(T \cap R) \times (T' \cap R') \subseteq H$.

and, for all states q, q' with $H(q, q')$, we have

- $\pi(q) = \pi'(q') \cap P$;
- For every set $T \in \delta(q, A)$, there exists a set $T' \in \delta'(q', A)$ such that for every set $R' \in \delta'(q', \Sigma' - A)$ there exists a set $R \in \delta(q, \Sigma - A)$ such that $(T \cap R) \times (T' \cap R') \subseteq H$.

[2] Our definition slightly generalises that of [3] by allowing multiple initial states and new propositions and agents.

The intuition is that whatever A can do in S, A can also do it in S' so that whatever the other agents do in S', they could already do it in S to yield a similar state. Intuitively, S' conserves all the capabilities A has in S, perhaps adding some more.

We say that S' A-simulates S, and write $S \leq_A S'$, if there is a simulation from S to S'. Intuitively, this holds if A has a superset in S' of the capabilities it has in S. It is proved in [3] that $S \leq_A S'$ iff every $\langle\langle A \rangle\rangle$-ATL* formula satisfied by S is also satisfied by S'. This formalises the intuition just mentioned, since formulas in $\langle\langle A \rangle\rangle$-ATL* assert capabilities of A.

3 Features and the feature construct

Our goal in this paper is to show how certain properties can be preserved through the addition of features. From this, we can demonstrate feature non-interaction, as explained in the introduction.

Our approach is to define a *feature construct* for the Mocha-like guarded command language introduced in section 2.4. The feature construct plays a similar role to the one defined for SMV [17]; it is also similar to the idea of superimposition [12]. Using it, we give examples of features and show, for specific features, that the system without the feature is an A-simulation of the system with the feature. From this, we conclude that properties of the base system are inherited by the system with features.

This section is structured as follows. In section 3.1 we model a Plain Old Telephone System (POTS) and some of its properties. Section 3.2 defines the feature construct, and gives some examples for POTS. We then study feature interactions in section 3.4.

3.1 POTS and its properties

Example 2. A more complete POTS model is defined using the guarded command language of section 2.4. In figure 2, we model the user: she may cause the phone to go offhook or onhook at will (**nondet** is a shorthand for a choice among all possible values of the type), and while the phone is offhook she may dial a number.

In figure 3, we model the exchange (without technical details). It consists of n identical agents, one for each user. It has a variable `st`, for status, which is initially `idle`. When the user goes offhook, `st` becomes `dialt`, for dialtone. If `st` is idle and another person tries to ring us, `st` becomes `ringing`, and we note the identity of the caller. If two users i, j simultaneously ring a third one k, the exchange must arbitrate by choosing one of them to succeed (gets ringing-tone) and the other one to fail (gets busy-tone). The exchange does this by setting `ex[`k`].caller` to i or to j.

The system consists of an array of exchanges and an array of users. Notice the parameter for EXCH: it is given the value of its own number, which it calls s (for 'self').

```
agent USER
controls
   offhook : boolean;
   dialed : Number;
init
   offhook := false;
update
   offhook -> dialed := nondet;
   -> offhook := nondet;
endagent;
```

Fig. 2. Code for USER

```
agent EXCH (s)
controls
   st : {idle, dialt, trying, busyt, ringingt, talking,
                              ringing, talked, ended };
   callee : Number;
   caller : Number;
init
   st' := idle;
update
   user[s].offhook & !user[s].offhook' -> st'=idle;
   st=idle & user[s].offhook' -> st' := dialt;
   st=idle & ex[j].callee=s &
                ex[j].st=trying & !user[s].offhook'
                -> st' := ringing; caller' := j;
   st=dialt & user[s].offhook'
                & user[s].dialed'=n -> callee' := n;
   ⋮
   st=trying & callee=j & ex[j].st=idle & ex[j].caller'=s
                       & user[s].offhook' -> st' := ringingt;
   st=trying & callee=j & ex[j].st=idle & ex[j].caller'!=s
                       & user[s].offhook' -> st' := busyt;
   st=trying & callee=j & !ex[j].st=idle
                       & user[s].offhook'-> st' := busyt;
   ⋮
endagent
```

Fig. 3. Code for EXCH

```
ex : array 1..n of EXCH;
ex[i] := EXCH(i);
user[i] := USER
```

Fig. 4. Code for POTS

The logic ATL is well-suited for expressing specifications of telephone systems, because the users are autonomous, and we are interested in whether they have the power to enforce certain behaviours. Compared with the properties defined using CTL in [17], ATL offers us the opportunity to distinguish between different sources of non-determinism, which makes the specification reflect our intentions more precisely. We illustrate with a few examples:

1. *Any phone may call any other phone.* In [17] this was approximated in CTL:
   $\forall i \neq j$. `A□E◇(ex[`i`].st=talking & ex[`i`].callee=`j`)`
   indicating that, in all reachable states, there is a path which eventually leads to i and j talking to each other. This is rather weaker than the intention, which was that it is within i's and j's joint power that i initiate a successful call to j. We may express that as $\forall i \neq j$
   `A□`$\langle\langle$`user[`i`],user[`j`]`$\rangle\rangle$`◇(ex[`i`].st=talking & ex[`i`].callee=`j`)`
   A similar formula which is slightly weaker but has the advantage of being within $\langle\langle \text{user}[i], \text{user}[j]\rangle\rangle$-ATL is $\forall i \neq j$:
   $\langle\langle$`user[`i`],user[`j`]`$\rangle\rangle$`□◇(ex[`i`].st=talking & ex[`i`].callee=`j`)`
2. *The user cannot change the callee without replacing the hand-set.* In [17] it is expressed in CTL as:
   ```
   A□ ((ex[i].callee=j & ex[i].st=trying)
               -> (A[ ex[i].callee=j W ex[i].st=idle ]))
   ```
   This is rather stronger than the intention: this forbids any change of callee. This CTL formula becomes false in the context of call-forwarding, where the system may change the callee as i sets up the call. In ATL, we capture the requirement more precisely:
   ```
   A□(ex[i].callee=j & ex[i].st=trying
               ->  [[user[i]]](ex[i].callee=j W ex[i].st=idle))
   ```
   This weaker formula is true even if the system can change the callee. Again, a slightly weaker formula in $[[\text{user}[i]]]$-ATL is possible:
   ```
   [[user[i]]]□(ex[i].callee=j & ex[i].st=trying
               ->  (ex[i].callee=j W ex[i].st=idle))
   ```

3.2 Feature construct definition

The feature construct that we use here is an adaptation of the generic idea of [17]. The base language that we use is a simplification of the *Reactive Modules* formalism [2] used by Mocha [1], that we presented in section 2.4.

Following [17], a feature can be seen as a prescription for changing a basic system. That which is assumed of a basic system will appear in the **require** section of the feature. Here, we can require particular agents and variables. The feature will add to the system new variables and agents to deal with the feature in the **introduce** section. Because many features need to be activated before taking effect, we usually introduce a boolean variable `use` that indicates whether the feature is activated. Finally, the **change** section indicates how the behaviour of the existing system is changed. Currently, we have four types of changes:

1. **if** *condition* **then override** c means that when *condition* is evaluated to true, the existing commands are disabled, and only the command c is allowed to execute.
2. **if** *condition* **then expand** c means that when *condition* is evaluated to true, the command c is allowed to execute. The existing commands are still enabled as before: the non-determinism of the system is increased.
3. **if** *condition* **then impose** c means that when *condition* is evaluated to true, the command c, which is a set of parallel assignments $x' := e$, determines the new values of these x variables. The values of other variables are set by an existing command.
4. **if** *condition* **then treat** c means that when *condition* is evaluated to true, the command c, which is a set of parallel assignments $x := e$, is used to determine the value of x in expressions. The variable x still exists, and will be accessible again when the condition reverts to false.

Only the last two types were present in [17]. The first two types can also be defined both in terms of syntactic manipulations or semantically, on the agent's transitions.

Finally a feature comes with **properties**, that describe its essential functionalities in a high-level way. These properties need not exhaustively specify the system. The specifier is intended to write properties which should be preserved when this feature is combined with other features. In this paper, we advocate the use of ATL* for properties.

Example 3. It is now very common to have many features on top of POTS. These features come in many variants, and are now being standardised [8].

For instance, the feature Call Forward When Busy (CFB) adds the following typical behaviour: When CFB is active and the subscriber's line is busy, incoming calls are diverted to a phone number pre-specified by the subscriber. The number can be changed, and the feature can be enabled or disabled at subscriber's will. The feature is implemented by changing the exchange of the caller, and adding new commands to the subscriber `i`, see fig. 5.

The fundamental property of forwarding is that user j can ensure that any user who tries to reach him will try user k instead, and j can choose any k. Note the scope of the quantifications (cf. remark 1.2).

Example 4. The feature Ring Back When Free (RBWF) also avoid the annoyance of busy callees, but this time it is a feature of the caller (me, say): If I get the busy tone when calling a number, I can activate RBWF. RBWF will then attempt to establish a connection as soon as the callee is free. It first calls me with a special ring; when I then lift the handset, a call is initiated on my behalf.

To model this (see fig. 6), we introduce `awaited`, the number we are trying to reach. Since we introduce a single number, only the last RBWF may be pending. Also we use Mocha's notion of **event** to model activation: it is

```
feature CFB(i)
require ...
introduce
  agent USER[i]
    controls
        use : boolean
        forw : Number
    init
        use := false
change
  agent USER[i]
    expand use := nondet;
    expand forw := nondet;

  agent EXCH(i)
   if st = trying & callee = i  & user[i].use & ex[i].st != idle
   then override callee' := user[i].forw;

properties
∀k.⟨⟨user[i]⟩⟩◇□∀j.(ex[j].st=trying & ex[j].callee=i & ex[i].st!=idle
   -> ex[j].st=trying U (ex[i].st=trying & ex[i].callee=k))
```

Fig. 5. Call Forward when Busy

an instantaneous action, whose occurrence can be caused by `event!` (equivalent to toggling `event`) and tested by `event?` (equivalent to the condition `event=event'`).

Let us have a closer look at the properties: the first one simply says that users together can make my ringback scenario succeed: I hear the special ringing, then I take the phone offhook and call `j`. The collaboration of all users is needed for this success:

- `i` must of course enable the feature.
- `j` must agree to be first busy, then idle.
- The collaborations of other users is needed as well, since they could conspire to hold `i` or `j` busy all the time.

The user `i` alone is much less powerful: He might decide not to use the feature at all, by not setting `activate`. (Indeed, the fact that the user can avoid using the feature is important to our main result, section 3.4.)

This leads to a natural categorisation of features, similar in motivation to [7], but different in detail: features can be categorised according to the set of players that occur in the cooperation modality of the ATL formula of their properties. This essentially says who is in control of the feature. Specifically, we can distinguish single-user features, two-users features, group features, system features (where system is a specific player).

```
feature RBWF(i)
require ...
introduce
 agent USER(i)
   event activate
 agent EXCH(i)
    controls
        use : boolean
        awaited : Number
        special_ring : boolean
    init
      use := false;
      special_ring := false;

change
 agent USER(i)
   expand activate!
 agent EXCH(i)
  if st = busyt & user[i].activate?
  then impose use' := true ; awaited' := dialed;

  if use & st = idle & ex[awaited].st = idle
  then override callee' := awaited;
                st' := ringing;
                special_ring' := true;

  if use & st = ringing & special_ring & user[i].offhook'
  then override st' := trying;
                  special_ring' := false; use' := false

properties
  ⟪user⟫ ◇((ex[i].st = ringing & ex[i].special_ring)
          U (user[i].offhook U ex[i].talking & ex[i].callee = j))
  ⟪user[i]⟫ !ex[i].use
  ⟪user[i]⟫◇((ex[i].use & ex[i].awaited=j) | (ex[i].st=ringingt))
```

Fig. 6. Ring Back When Free

3.3 Feature Construct Semantics

We define the semantics of the feature constructs **override**, **impose**, **expand** and **treat** by syntactic transformation of the Mocha-like language. Dealing with the **require** and **introduce** sections is straightforward: for **require**, we check that the required items are present (the feature integration fails if they are not), and for **introduce** we simply add the new data.

The **change** section is dealt with as follows. Suppose we start with the program in figure 7, and we integrate a feature.

```
agent A
controls
 ...
init
 ...
update
   g1 -> c1;
   g2 -> c2;
   ⋮
   gn -> cn;
endagent
```

Fig. 7. Some arbitrary code for an agent A.

- For the feature **if** g **then impose** $x := e$, the update section of the program becomes:

```
g1 & !g -> c1;
g1 & g -> c1 [x:=e];
g2 & !g -> c2;
g2 & g -> c2 [x:=e];
  ⋮
gn & !g -> cn;
gn & g -> cn [x:=e];
```

  The meaning of $c[x := e]$ where c is a set of assigments is to replace (if present) the assigment of x in c by the new one $x := e$, or to add it (if not present). (Recall that in Mocha the list of assignments are performed simultaneously.)
- For the feature **if** g **then override** c, the update section of the program becomes:

```
g1 & !g -> c1;
g2 & !g -> c2;
  ⋮
gn & !g -> cn;
g -> c;
```

- For the feature **if** g **then expand** c, the update section of the program becomes:

```
g1 -> c1;
g2 -> c2;
  ⋮
gn -> cn;
g -> c;
```

- For the feature **if** g **then treat** $x = f$, the update section of the program becomes:

```
g1 -> c1';
g2 -> c2';
  ⋮
gn -> cn';
```

  where x'_i is c_i but with x replaced with the conditional expression $g?f : x$ (i.e. if g then f else x).

3.4 Feature interactions

Thanks to the properties that are part of our features, we can define interactions as a discrepancy between the expected properties of the system with features and the actual ones. We note a feature as (F, ϕ) where ϕ is the properties sections and F is the description of how the feature is implemented. Applying the feature F to a system S satisfying its requirements will be denoted $S + F$. This operation is also called "feature integration". We assume that the **require** section, the **change** section, and the introduced properties are consistent with each other: that is, that $S + F \models \phi$ for any S satisfying the requirements.

Now we can define a feature interaction as non-preservation of the properties of integrated features:

- The feature F interacts with the system S by destroying some core property ϕ_S of the system: $S + F \not\models \phi_S$;
- The feature F_2 interacts with the feature F_1 by destroying a property ϕ_1 introduced by F_1: $S + F_1 + F_2 \not\models \phi_1$.

The goal of feature-oriented programming is to be able to produce rapidly systems with a large number of features integrated, and to ensure the absence of feature interactions for such systems.

It is thus important to prove generic preservation properties: a feature F preserves all properties of a class C if (for all $\phi \in C$) $S \models \phi$ implies $S+F \models \phi$.

We have seen that simulation relations are the right tool to this end: they ensure that a wide class of properties are preserved when adding a feature. These relations give a precise meaning to the notion of backward compatibility.

In particular, if we can show that $S + F$ A-simulates S (for any S), when integrating F in a new system, we know that many properties of this new system do not need to be checked. Features are usually intended to augment the power of their users: formulas talking about these powers are thus preserved.

Proving this property of F can sometimes be done easily. First, we define an *A-enabled variable* `use` *introduced by* F (where $A \subseteq \Sigma$) to have the following properties:

- the variable is **introduce**d by F in some agent a
- the variable is initially false: `use:=false` appears in an **initintroduce**d by F.
- the variable can only be set by agents in A using **expand**ed commands. This can often be checked syntactically: For instance, if `use` is only set to true by an **expand**ed command of an agent in A, as in `CFB`, this is immediate. In `RBWF`, this is indirect: `use` is controlled by `exch[i]` (an agent outside A) but the guard contains a variable `activate` controlled by an **expand**ed command of user[i]. More generally, this can be verified by checking that the feature F', which is F without the **expand** of agents in A, satisfies A$\Box\neg$`use`, by which we mean that for any base S, $S + F' \models$ A$\Box\neg$`use`.

This condition can be used to ensure that, if agents in A behave exactly as they did in the old system, they will not enable the feature.

Theorem 1. If all changes of F are of one of the following forms:

- a change that is guarded by an A-enabled variable introduced by F
- an **impose** where all affected variables are **introduce**d by F.
- an **expand** of an agent in A.
- "**if** $g \wedge g'$ **then override** c" in an agent in $\Sigma - A$, if $g \rightarrow c$ is a command of this agent.

then F is A-preserving, i.e. $S + F$ A-simulates S for any S that satisfies the **require** clause.

The idea of the proof is to note that the relation obtained by requiring that all old variables have the same value, and that the enabling variables `use` are false, is an A-simulation from S to $S + F$. Thus all properties written in $\langle\langle A \rangle\rangle$-ATL are preserved. We cannot give a real proof of this theorem here, as it requires the precise semantics of our Mocha-like language which is not given in this paper.

Example 5. The features `CFB(i)` and `RBWF(i)` are A-preserving for any set of agents A containing user[i]. Since their properties are also in this fragment, these features will not interact (in the sense of this paper).

Note that it is usually considered that these features do interact, since a user A that called B, was forwarded to C and activated RBWF might well

end up calling back C in some implementations, while he probably intended to call B. Here this interaction is correctly, but silently, handled by our model, since it does not belong to the class of interactions defined in this paper.

3.5 Preliminary Experiments

Currently, there is no automatic translation from our Mocha-like language to Mocha. However, we have successfully implemented the model of the POTS given in Fig. 2 to 4 of section 3.1 with 4 users, and checked the properties discussed in section 3.1.

4 Conclusions

We have shown a general case in which introducing a feature provably does not break a class of properties: this holds when integrating the feature results in a U-simulation of the original system for some group of users U, and the properties assert capabilities of the users in U. We have indicated four types of changes that are U-preserving. We illustrated with examples from the telephone system. Most telephone features naturally fit into one of the cases of the theorem. Thus the proofs of non-interaction that [17] had to perform for all combination of features can now be obtained by a simple, single syntactic check.

The general technique, in principle, can work for any logic and its associated notion of simulation. However, we have found that ATL* provides a rich set of fragments and associated simulations, that are suited to the application domain: features are valuable only because they offer new capabilities to their users, and thus their properties are naturally expressed in ATL*. Actually, our example properties were all in the smaller fragment $\langle\langle A\rangle\rangle$-LTL[3], for which the weaker $\langle\langle A\rangle\rangle$-trace containment suffices. We didn't pursue this line of research since all our features happen to be preserving also the stronger $\langle\langle A\rangle\rangle$-simulation, and this preservation is easier to show.

The special case where $U = \emptyset$ allows to show the preservation of invariants of the system (or more generally, ACTL* formulas). However, the corresponding simulation only allows features to make their agents more deterministic, which is rarely useful.

We have seen intuitively appealing properties of the form $\mathrm{A}\Box\langle\langle U\rangle\rangle\Diamond\phi$. Our method could be extended by discovering the "simulation" relation corresponding to these formulas, and looking for a simple way to prove that a feature preserves this relation. We plan to define a suitable notion of U-*resettable* systems. Intuitively, the telephone system is resettable by its users: if they all hang up and switch off their features, the system returns to its initial state. We would like to define this precisely, and prove of U-resettable systems that $\langle\langle U\rangle\rangle\Diamond\phi$ is equivalent to $\mathrm{A}\Box\langle\langle U\rangle\rangle\Diamond\phi$ (from left to right is done

by prefixing the strategy with a reset). This would imply that formulas of the form $\mathrm{A}\Box\langle\langle U\rangle\rangle\Diamond\phi$ are preserved by U-conservative features.

The idea of this paper may be seen as a special case of a proof rule of the form

$$\frac{S \models \phi \qquad \text{condition on } F, \phi}{S + F \models \phi}$$

which allows us to preserve the property ϕ through the addition of the feature F. In this paper, the condition on F, ϕ is that F is U-conservative and ϕ is in $\langle\langle U\rangle\rangle$-ATL*. Other conditions on F, ϕ can be used. In another paper, we are modelling features as warps in the transition system and deriving from ϕ a simpler formula which the warp is required to preserve [11].

A related problem is to show the internal consistency of features, by which we mean that $S + F \models \phi$ for any S that satisfies the requires clause. By inserting the needed properties in the requires clause, the combination of features could eventually become a matter of plug and play, with well defined and easily combinable compatibility properties.

Finally, we used here only two levels for describing a property: the level of models, and of formulas. Lower levels indicating how to integrate features at the level of code would make the approach practical, and checking consistency between levels will improve our confidence in features.

Achkowledgments. The three authors are members of the FIREworks[3] Esprit Working Group, and gratefully acknowledge support for travel which enabled them to meet together. Mark Dermot Ryan also acknowledges British Telecom for generous support, and Pierre-Yves Schobbens thanks the University of Birmingham for funding an invited professorship that provided a further opportunity to work on this material.

References

1. R. Alur, H. Anand, R. Grosu, F. Ivancic, M. Kang, M. McDougall, B.-Y. Wang, L. de Alfaro, T. Henzinger, B. Horowitz, R. Majumdar, F. Mang, C. Meyer, M. Minea, S. Qadeer, S. Rajamani, and J.-F. Raskin. *Mocha User Manual.* University of California, Berkeley. `www.eecs.berkeley.edu/~mocha`.
2. R. Alur and T. Henzinger. Reactive modules. *Formal Methods in System Design*, 15(1):7–48, 1999.
3. R. Alur, T. Henzinger, O. Kupferman, and M. Vardi. Alternating refinement relations. In D. Sangiorgi and R. de Simone, editors, *CONCUR 98: Concurrency Theory*, Lecture Notes in Computer Science 1466, pages 163–178. Springer-Verlag, 1998.
4. R. Alur, T. A. Henzinger, and O. Kupferman. Alternating-time temporal logic. In *Proceedings of the 38th Annual Symposium on Foundations of Computer Science*, pages 100–109. IEEE Computer Society Press, 1997.

[3] `www.cs.bham.ac.uk/~mcp/fireworks/`

5. M. Calder and E. Magill, editors. *Feature Interactions in Telecommunications and Software Systems VI*. IOS Press, 2000.
6. M. Calder and S. Reiff. Modelling legacy telecommunications switching systems for interaction analysis. In *Systems Engineering for Business Process Change*. Springer Verlag.
7. E. Cameron, N. Griffeth, Y.-J. Lin, M. Nilson, W. Schnure, and H. Velthuijsen. A feature interaction benchmark for in and beyond. In W. Bouma and H. Velthuijsen, editors, *Feature Interactions in Telecommunication Systems*. IOS Press, 1994.
8. CCITT. *Recommendation Q.1215, Distributed Functional Plane for Intelligent Network CS1.*, 1992.
9. E. M. Clarke and E. A. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In D. Kozen, editor, *Logic of Programs Workshop*, number 131 in LNCS. Springer Verlag, 1981.
10. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.
11. H.-D. Ehrich, M. D. Ryan, and P.-Y. Schobbens. Preserving temporal properties through time warps. In preparation.
12. S. Katz. A superimposition control construct for distributed systems. *ACM Transactions on Programming Languages and Systems*, 15(2):337–356, April 1993.
13. K. Kimbler and L. G. Bouma, editors. *Feature Interactions in Telecommunications and Software Systems V*. IOS Press, Sept. 1998.
14. M. Kolberg, E. Magill, D. Marples, and S. Reiff. Results of the second feature interaction contest. In Calder and Magill [5], pages 311–325.
15. H. Korver. Detecting feature interactions with CÆSAR/ALDÉBARAN. *Science of Computer Programming*, 29(1–2):259–278, July 1997.
16. K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.
17. M. Plath and M. D. Ryan. Entry for FIW'00 Feature Interaction Contest. Technical report, School of Computer Science, University of Birmingham, 2000. Available from `www.cs.bham.ac.uk/~mdr/papers.html`. Also summarised in [?].
18. M. C. Plath and M. D. Ryan. Feature integration using a feature construct. *Science of Computer Programming*, 2000. To appear. A shorter and earlier version of this paper appeared in [13], pages 150–164.