DOI 10.1007/s00165-008-0101-8
BCS © 2009
Formal Aspects of Computing (2009) 21: 571–588

Formal Aspects
of Computing

# Model-checking user behaviour using interacting components

Thomas Anung Basuki, Antonio Cerone, Andreas Griesmayer, Rudolf Schlatte

International Institute of Software Technology, United Nations University, Macau SAR, China.
E-mail: anung@iist.unu.edu

**Abstract.** This article describes a framework to formally model and analyse human behaviour. This is shown by a simple case study of a chocolate vending machine, which represents many aspects of human behaviour. The case study is modelled and analysed using the Maude rewrite system. This work extends a previous work by Basuki which attempts to model interactions between human and machine and analyse the possibility of errors occurring in the interactions. By redesigning the interface, it can be shown that certain kinds of error can be avoided for some users. This article overcomes the limitation of Basuki's approach by incorporating many aspects of user behaviour into a single user model, and introduces a more natural approach to model human–computer interaction.

**Keywords:** Human–computer interaction; Interacting components model; Rewrite systems; Model-checking

## 1. Introduction

Formal methods and user-centred design are two alternative methodologies aiming at reducing the likelihood of system failure. In user-centred design analysing and foreseeing user's behaviour in interacting with the computer, as well as possible human errors, is a fundamental concern of designers, who also have to test the validity of their assumptions in the real world, with actual users. Formal methods have been originally applied to system design and analysis with the perspective that human errors are outside their scope.

However, this perspective started gradually to change during the 1980s. Firstly, more and more importance is given to the formal analysis of user interfaces. Chi [Chi85] compares four algebraic techniques in the analysis of a commercial user interface, manually proving a number of not trivial properties of the interface, which however do not involve the interaction with the user. Secondly, it is understood that a user model should be included in the design process, separately from the model of the interface. Cognitive complexity theory (CCT) [KP85] clearly separates the description of the user's goals from the description of the device with which the user interacts. Another interesting approach is the Executable Cognitive Architecture developed by Newell, Laird and Rosembloom, which was then implemented in the SOAR system [LNR87]. This cognitive architecture inspired the definition of the programmable user model (PUM) defined by Young, Green and Simon [YGS89], which envisage an executable model of the user within a psychologically constrained architecture which allows a predictive evaluation of the interface design.

In the 1990s, the catastrophic consequences of human errors experienced in safety-critical systems, including for example plant/process control, traffic control (air, road, rail, sea), medical devices and defence, gave a

*Correspondence and offprint requests to*: T. A. Basuki, Email: anung@iist.unu.edu stronger impulse to the application of formal methods to interactive systems [Dix91]. The aim is no longer just modelling, but becomes the application of formal verification techniques, such as model-checking and theorem proving, to the verification of interactive systems. Works in this direction include the model-checking analysis of interfaces in absence of user models [PS01, LH03, DCH97, DT04], and the description of expected effective user behaviour [Lev97, PBP97] or errors performed by the user as reported by accident analysis [Joh97].

However, the emergence of the ubiquitous computing paradigm has modified the traditional relationships between users and computational services by extending the computational interface into the user's environment. Providing computational services throughout the physical environment without making the user aware of the presence of the computer requires extreme rigour in the application of principles of user-centred design and needs to be supported by the study of human errors. The informal user-centred design approach appears insufficient when computational services are concealed within everyday objects and activities, which are freely manipulated, often in a physical sense, by a wide range of users. Analysing and foreseeing the behaviour of users with so many degrees of freedom and covering all ranges of expertise, including novices, becomes unpractical. The use of formal methods becomes then a very attractive option, but their scope needs to be broadened compared to the early applications to interactive systems. In fact, users do not necessarily behave as expected while designing the interface, as assumed in previous work [Lev97, PBP97, Joh97], and errors are actually the very result of an unexpected user behaviour that emerges during the interaction.

To best capture such an emergent behaviour, we definitely need an explicit model of the user, which, however, must specify the cognitively plausible behaviour, which is all the possible behaviours that can occur, and that involve different cognitive processes [BBD00]. The model must take into account all relationships between user's actions, user's goals and the environment. Following this approach, a number of researchers have explored the use of formal models to understand how cognitive errors can affect user performance. Mode confusion is often blamed for human errors in interactive systems, whereby the user's mental model of how the system works gets out of step with how the system is actually working. This phenomenon has been analysed by developing a plausible (but simplified) mental model such as might represent the user's understanding of the system and comparing it with a user interface model, to look for sources of cognitive dissonance [BMPC98, Rus02]. Rushby [Rus02] models the behaviour of a forgetful user who follows a warning display light or a non-forgetful user working in absence of warning lights, and checks for emergent mode confusion. Curzon and Blandford [CB00b, CB00a, CB01] focus on goal-based interactions and use a chocolate machine case study to build a rational user model and formally prove the absence of several classes of user errors:

- assuming all tasks completed when the goal is achieved, but forgetting to complete some important subsidiary tasks (post-completion error);
- given a list of communication-goals that must be communicated to the machine, without knowledge of the right order of communication permitted by the machine, performing information communication in a wrong order (order error);
- leaving the interaction whenever a delay occurs and no feedback is given by the machine (device-delay error).

Cerone and Elbegbayan [CE06] model a non-expert user interacting with a web-based application. They use the CSP process algebra [Hoa85] and the CWB-NC model-checker [CLS00] to detect user errors related to security issues and progressively improve the interface design.

Basuki [Bas07] formalises a variant of Curzon and Blandford's chocolate machine case study within Cerone and Elbegbayan's framework and models the interaction of two kinds of users, goal-based user and reactive user, with the machine. He uses temporal logic to define all different classes of errors considered by Curzon and Blandford [CB00b, CB00a, CB01]. The CWB-NC model-checker is then iteratively used to find the cause of a specific error and improve the design to remove that error. Basuki uses a goal-based user as a general user model and applies a constraint to it through the parallel composition operator of the process algebra in order to define a reactive user. However, the addition of a constraint to a process algebra model reduces the set of possible system behaviours, and in Basuki's case study leads to the loss of the goal-based part of the user behaviour. As a result, different behavioural attitudes that normally coexist within the same user behaviour are modelled in Basuki's approach as separate users: a goal-based user and a reactive user. Therefore Basuki's approach cannot be applied to more complex case studies in which goal-based attitude and reactive attitude are both essential in the completion of the same user's task.

In this article, we overcome the limitations of Basuki's approach by using interacting components to model human–computer interaction, and show how to implement it by using the Maude rewrite system [CDE+07] to model the chocolate machine case study. With the new framework, rather than adding a constraint for each new set of user behavioural attitudes, we add new rules to the user model. In this way, the existing behaviour can coexist with the newly added one. A rewrite system is suitable for implementing these ideas, because adding new set of rules to refine a user model in a rewrite system is easier than in a process algebra. In a rewrite system, we can start by defining a simple user model, and refine the model by adding more rewrite rules. On the contrary in process algebra we will have to start with the complex user model from the beginning.

## 2. Interacting components model

In this section we show a way to model multiple components ("peers") involved in human–computer interaction. Human actors and machine actors are both peers. In addition to the interacting peers, we explicitly define an interface component for communication among the peers. The interface allows us to describe the peers independently of each other and to concentrate on realistic models of different kinds of users and machines. Without using the interface, the two models of user and machine behaviour would have to be combined by hand. Peers do not communicate with each other directly. They have their own internal states and can perform their own internal actions. Communications between peers are performed through the interface by actions that affect the state of the interface. This change of state can be observed by other peers and trigger local actions. For example, to model a customer inserting a coin into a chocolate selling machine, we have a rule that simultaneously decreases the number of coins of the customer and "fills" the coin slot of the interface. This rule only involves the customer and the interface. A second rule, involving only the interface and the chocolate machine itself, removes the coin from the interface and increments the number of coins held by the machine.

We represent the components using labelled transition systems (LTS) and distinguish between the interface $I = \langle \Sigma_I, \rho_I, S_I \rangle$ and the set of peers $P_k = \langle \Sigma_{Pk}, \rho_{Pk}, S_{Pk} \rangle$ with $0 \leq k < n$ for $n$ peers and $\Sigma$ being the alphabet, $\rho$ the transition relation and $S$ the disjoint set of states. The literals in the alphabet resemble actions the peers can perform. We abbreviate the set of all actions as $\hat{\Sigma}_P = \bigcup_{0 \leq k < n} \Sigma_{P_k}$. Communication between the peers can only occur through a change of state in the interface $I$. Moreover, the state of the interface may change only through a synchronisation with one of the peers. Thus $\Sigma_I \subseteq \hat{\Sigma}_P$. We write $s_i$ for states in $S_I$ and $s_{p_k}$ for states in $S_{P_k}$. Combination of all components in respect to the interface gives us the total system $M = \langle \Sigma_M, \rho_M, S_M \rangle$ as follows:

$$\Sigma_M = \hat{\Sigma}_P \tag{1}$$

$$S_M = S_I \times S_{P_0} \times \cdots \times S_{P_{n-1}} \tag{2}$$

$$\begin{aligned} \rho_M = \{((s_i, s_{p_0}, \ldots, s_{p_k}, \ldots, s_{p_{n-i}}), \sigma, (s_i', s_{p_0}, \ldots, s_{p_k}', \ldots, s_{p_{n-i}})) \mid \\ \sigma \in (\Sigma_I \cap \Sigma_{P_k}) \wedge (s_i, \sigma, s_i') \in \rho_I \wedge (s_{p_k}, \sigma, s_{p_k}') \in \rho_{P_k}\} \end{aligned} \tag{3}$$

$$\begin{aligned} \cup \{((s_i, s_{p_0}, \ldots, s_{p_k}, \ldots, s_{p_{n-i}}), \sigma, (s_i, s_{p_0}, \ldots, s_{p_k}', \ldots, s_{p_{n-i}})) \mid \\ \sigma \notin \Sigma_I \wedge (s_{p_k}, \sigma, s_{p_k}') \in \rho_{P_k}\} \end{aligned} \tag{4}$$

A state $s_i \in S_M$ in $M$ represents the internal states of all participants. The set of transitions $\rho_M$ consist of the union of two parts: Set (3) represents interface actions, which change the state of the interface and of one peer at the same time. They represent the means of communications between the peers. This form of communication is asynchronous: one peer changes the state of the interface to a state where another peer can continue. To stay in our running example: the chocolate machine can change the state of the interface by putting a coin into the return slot, which in turn can be removed by the user. After these two steps the interface is in the same state as before, while the machine progresses to a state that reflects that the change was given and the user has received the return money. Set (3) describes state changes by actions that are not part of the interface. Note that each transition changes exactly the state of a single peer; there are no direct synchronisations between peers.

It is easy to see that the combined system $M$ is able to perform exactly the intersection of the behaviour of the peers, synchronised by the interface. In other words, $M$ models the interaction between the peers.The complete

```
mod CHOC-MACHINE is
   inc INTERFACE .

   sort MState .

   ops Ready CoinAccepted ChocSelected Delayed ChangeGiven Stopping : -> MState [ctor] .

   op State:_ : MState -> Attribute [ctor] .
   ops Euro:_ 50cent:_ Choc:_ : Nat -> Attribute [ctor] .
   .
   .
   .
```

**Fig. 1.** The static parts of the machine model. The `Attribute` constructors "wrap" terms of sorts `MState` and `Nat`, respectively

system resembles a labelled transition system and therefore is amenable for model checking. Accordingly, we state the properties to check as LTL formulae.

*Linear temporal logic* (*LTL*) [Pnu77, CGP99] is a temporal logic built from the actions in $\Sigma$, their negations, the temporal modalities $X$ (next), $U$ (until), and $R$ (releases), and Boolean conjunction and disjunction. An LTL formula defines a set of words. Intuitively, $X\phi$ holds for a given word if $\phi$ holds in the suffix of the word starting from the second position, $\phi\, U\psi$ holds if $\psi$ holds on some position and $\phi$ holds on every position before that point, and $\phi R\psi$ holds if either $\psi$ holds on all positions, or $\phi \wedge \psi$ holds on some position and $\psi$ hold in all positions before. We use the usual abbreviations $\Diamond\varphi = \mathit{true}\ U\varphi$ (eventually) and $\Box\varphi = \mathit{false}\ R\varphi$ (always). Intuitively, $\Diamond\varphi$ states that any path in the model eventually will reach a state in which $\varphi$ will be true, while $\Box\varphi$ states that $\varphi$ is true now and for all reachable states in the model.

## 2.1. Implementation

The model above can easily be implemented in any formalism supporting LTS or automata. A number of such formalisms and tools exist for, e.g. designing hardware or as input languages for model checking tools. A well suited formalism used in previous work [Bas07] is CSP [Hoa85]. Most of these tools, however, are centered on the development of single components (processes), which then are explicitly synchronized. The events for synchronization have to be chosen carefully and kept consistent while implementing the single components. A more direct way for modelling in our setting is to concentrate on the actions and implement the transitions given in Formula (3) and (3) directly.

The formalism most suitable for the approach in this paper is the rewriting system Maude, which allows us to model the transitions for the actions in $\Sigma$ directly. The flexibility of Maude also allows us to adjust the syntax to simplify modelling. Maude [CDE+03] is a language based on equational and rewriting logic. It can be used to model systems in an algebraic style (via equations) as well as in a state-based fashion (via rewrite rules). Maude also supports an object-oriented programming paradigm and model-checking capabilities. We only show the subset of the language that is necessary for our modeling purposes; for further information, see the Maude manual [CDE+07].

Maude code is organized in *modules*, which may import and/or extend other modules. Contained inside a module are *sort definitions*, *operators* that are used to construct *terms*, *equations* specifying term equivalences and *rewrite rules* describing state changes.

An example that shows the syntax of Maude is given in Fig. 1. Module `CHOC_MACHINE` is defined by including module `INTERFACE` (keyword `inc`), defining sort `MState` (keyword `sort`) and several operators (keyword `op` for single and `ops` for multiple definitions), which are all *constructors* (keyword `ctor`), which means that their occurrence creates an instance of sort `MState` or `Attribute`, respectively. The `Attribute` constructors take one argument each, so each attribute will contain a value of the specified sort; the `MState` constructors do not take an argument. The sorts not explicitly defined in this and subsequent examples (`Cid`, `Attribute`,...) are pre-defined by Maude and directly or indirectly inherited in the modules we define.

The start of the `CHOC-MACHINE` module, shown in Fig. 1, defines the attributes necessary to give the state of the chocolate machine. The constructors `Euro:`, `50cent:` and `Choc:` represent data attributes and hold the amount of money and items (chocolate) in store. All these data attributes are constructed from a value of type `Nat`, a natural number. In addition to the data attributes, we use a state machine to store the state of interaction the machine is in. This state machine has the control states`Ready`,`CoinAccepted`,`ChocSelected`,`Delayed`,`ChangeGiven`

```
rl [ acceptcoin ] :
     < int : Interface | InCoin: true, CoinLight: true, ChocLight: false,
                         ChocButton: C, A >
     < machine : Machine | Euro: N, State: Ready, A' >
  =>
     < int : Interface | InCoin: false, CoinLight: false, ChocLight: true,
                         ChocButton: false, A >
     < machine : Machine | Euro: N + 1, State: CoinAccepted, A' > .
```

**Fig. 2.** Rewrite rule for receiving a coin through the interface. Unused attributes are subsumed in the attribute lists `A` and `A'` and left unchanged by the rule

and `Stopping`, which are given as constructors for the sort `MState`. The constructor `State:` produces an attribute to hold the machine state.

The rewriting part of Maude uses rewrite rules to rewrite a term to a different term of the same sort. An example of a rewrite rule is given in Fig. 2. We see two objects representing the interface and the machine on the left hand side of the rewrite rule and the corresponding objects as result of a rewrite, where `Interface` and `Machine` are the classes of the objects, and `int` and `machine` are the unique object identifiers (`Oids`). We do not need to reason about the actual names; in fact, the occurrence of the same identifier in the left- and right hand side ensure that both sides refer to the same object. The set of attributes that are not of concern for the rewrite rule (like the number of chocolate bars still present in the machine) are contained in variables `A` and `A'` and are not changed by the rewrite rule. The objects are changed synchronously, if there is a coin in the interface (`InCoin` = *true*), the control state of the machine is changed from `Ready` to `CoinAccepted` and the number of Euro coins is increased by one (`Euro` attribute).

We implement interacting component models in terms of rewrite rules. All rewrite rules for a peer $P_k$ are grouped in one Maude module. Each rewrite rule relates to an action $\sigma$ and changes the state of $P_k$ and optionally the state of the interface. A state is defined in terms of *attributes*. An attribute can represent a control state as well as data values. Although not relevant for computation, this distinction allows for better understanding of the meaning of a model and simplifies the modelling process.

It is easy to see that each of the modules implements the transition relation $\bar{\rho}_k \subseteq (s_i \times s_{p_k}) \times \sigma \times (s'_i \times s'_{p_k})$, with $s_i = s'_i$ for actions that do not involve the interface. A rewrite rule is performed atomically, and states not changed by the rewrite rule remain unchanged. Therefore

$$\rho_M = \{((s_i, s_{p_0}, \ldots, s_{p_j}, \ldots, s_{p_{n-i}}), \sigma, (s'_i, s_{p_0}, \ldots, s'_{p_j}, \ldots, s_{p_{n-i}})) \mid ((s_i, s_{p_j}), \sigma, (s'_i, s'_{p_j})) \in \bar{\rho}_{P_j}\}$$

This modular structure allows for exchanging single components by local changes to a single module. Therefore, we can easily check the effects of changing the user behaviour by adjusting the rules for the user and its interaction with the interface which are in the same module. The description of other peers (the machine) can be left unchanged.

## 3. Modelling human–computer interaction

In this paper we have two goals that we intend to achieve. First, we intend to explore the capability of interacting components model to model human–computer interaction. The focus is on user models. We try to incorporate within a single user model all aspects of user behaviour that in Basuki's work [Bas07] were actually formalised by distinct user models. The second goal is to use model-checking to verify a machine interface design. Here the focus is on the interaction between human and machine.

This section is divided into two parts. The first part of this section describes some general properties of user behaviour that we want to model. The second part of this section describes how we model these properties using the interacting components model. In Sect. 4 we use the chocolate machine case study as an example of how we implement our approach. In Sect. 5, we discuss how we use model-checking to analyse our model.

### 3.1. User behaviour

When we try to model a user interacting with a machine, it is necessary to make assumptions on the way the user behaves. In this paper we try to model a rational user. When interacting with a machine, a rational user aims to achieve a goal. The user will try to perform actions in order to achieve the goal. This means that we exclude random user behaviour from our model.

In this section we give some definitions that characterise specific aspects of the user behaviour.

**goal-based** A goal-based user aims to achieve a *goal* through an interaction with the system. The user is likely to leave the interaction after achieving the goal.

**communication goals** They are the pieces of *information* that the user has to communicate to the machine in order to achieve the intended goal.

**reactive behaviour** The user decides which action to take in accordance to a set of stimulus-response pairs. The user selects a response that is associated with the perceived stimulus depending on the current state of the interaction.

**user habituation** After interacting with the same machine for several times, a user may become *habituated* with the machine behaviour. The user knows a pattern of machine behaviour, as a response to a sequence of user actions. A habituated user may wish to make use of such knowledge about the machine behaviour to achieve the goal faster. Since the user knows the order of actions to perform and the response the machine will give, actions can be performed without waiting for the stimuli given by the machine, hoping the machine will respond faster so that the goal can be achieved earlier. If this is allowed by a specific machine, it may lead the user to wrong conclusions, thinking that every machine behaves in that way.

**impatience** During the interaction with a machine, a user has a tolerance time. Obviously different users may have different tolerance times. The user is aware of the need to wait for the machine to finish processing and is prepared to wait provided that the processing time is not longer than the tolerance time. If the processing time is longer than the tolerance time, the user may suspect that something is wrong with the machine and may either redo a previous action or leave the interaction.

**carefulness** We model users who are careful with their possessions. They do not want to lose any of their possessions without achieving any goal. In an interaction with a machine, careful users are only willing to give one of their possessions to the machine in order to exchange it with another thing that they wish to get from the machine. In this case the value of the thing they get from the machine is considered equal to the value/price they have to give away, so that the value of their possessions remains unchanged. Carefulness affects users' decision when they have to redo an action. Careful users will only consider to redo actions when they believe that something has gone wrong with the machine. In this case they suspect that one of their previous actions has failed. Careful users will only redo actions that still preserve the value of their possessions.

### 3.2. Modelling user behaviour

This subsection explains how we model user behaviour using the interacting components model. We define the computer and the human user as components with attributes. The attributes and rewrite rules are used to model all characteristics of user behaviour as listed in Sect. 3.1.

**goal-based** We define an attribute in the user model to indicate whether the user has achieved the goal. We use rewrite rules to define how the user leaves the interaction after achieving the goal. In all other rewrite rules for the user, we include the condition that the user's goal has not been achieved as preconditions.

**communication goals** In our approach we model user's mental state as one of the user's attributes. The user's mental state indicates which communication goals have been communicated to the machine.

**reactive behaviour** To model reactive behaviour, we must define stimuli as attributes in the interface and define rewrite rules that define how the user react upon the stimuli.

**user habituation** We model habituation by using rewrite rules describing user actions based on the user's mental state. These rewrite rules ignore stimuli from the interface, and take user's mental state as preconditions. In this way we can model a user that is habituated with a kind of interaction pattern and does not react on machine's stimuli.

**impatience** To model user impatience, we must first model user's tolerance time. Since we work with discrete formalism and tool, we face a problem to model the tolerance time, which is continuous. Therefore, instead of modeling exact time, we define an attribute in the interface, that represents the user observation of device delay. Time delay is now modeled by a rewrite rule that is non-deterministically taken by the machine and sets the device-delay attribute. Finally we define rewrite rules that have the device-delay attribute as preconditions, representing user reactions to the device-delay. With these rules we can model both impatient users, who will leave interaction or redo an action upon device delay, and patient users, who will wait until the machine continues with the next action.

**carefulness** In our approach we model a careful user who only wants to redo actions which do not involve giving some of its possessions to the machine. We model this phenomenon by defining rewrite rules for redoing an action when the user believes that its previous action has failed (which is indicated by device-delay).

## 4. Chocolate machine case study

To demonstrate the applicability of our approach, we model the interaction between a user and a vending machine selling chocolate. Although this is a simple case study, it allows us to show many typical kinds of human–computer interactions and errors in interface design. To keep the model understandable, while still being realistic, we introduce some simplifications that reduce the state space, but preserve the main behaviour. Therefore, the user is required to select the item to buy although the machine only provides one kind of item (chocolate bar), and the machine always gives the same amount as change.

In this section, we introduce the model of the chocolate machine interface, a basic model of the vending machine itself, and a model of the user who wishes to buy a chocolate bar. In Sect. 5, we will demonstrate how to check for correctness of the system, and how to improve the machine in terms of usability.

### 4.1. The interface model

Figure 3 shows a graphical representation of the user interface of our vending machine. The interface module in Fig. 4 follows the physical interface. It models the six visible items on the machine: two lights (`CoinLight` for coin insertion and `ChocLight` for chocolate selection), one button (`ChocButton`), and three slots (`InCoin`, `OutCoin` and `OutChoc` slots). The lights can be turned on by the machine to instruct the user to perform actions. The `InCoin` slot is the place where the user can insert (and the machine can remove) a 1 € coin. The `OutCoin` slot is the place for the user to take the 50 cent coin, after the machine has placed it there. Similarly, the `OutChoc` slot is the place from where the user can take the chocolate bar.

In addition to the physical features, the Maude model of the interface also encompasses other, more intangible features of the machine. The `MchDelay` attribute does not represent any object that is visible on the interface, but rather models the user observation of a device delay which is needed by the machine to process information received from the user. If its value is `true`, it represents the case that the user can observe a device delay, otherwise its value is `false`.

### 4.2. The chocolate machine model

We give an initial model of a simple machine that only allows the user to perform actions in a strict order. The machine will first accept the coin from the user. Then it will accept the chocolate selection. The user cannot perform actions in a different order. The machine uses the coin light and chocolate light of the interface to indicate which action can be accepted. If the user performs an action before the machine is ready to accept it, this action will be ignored by the machine. We have the following control states:

**Ready** the machine is ready to accept coins

**CoinAccepted** the machine has accepted one coin and is ready to process chocolate selection

**ChocSelected** the machine has processed chocolate selection

**ChangeGiven** the machine has given a coin as change

**Stopping** the machine has stopped working because it has no more chocolate bars or it has no more 50 cent coins.

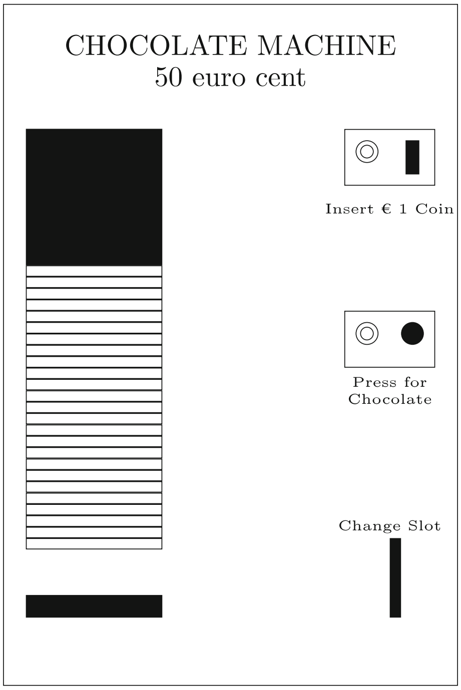

**Fig. 3.** The chocolate machine

```
mod INTERFACE is
    pr NAT .
    including CONFIGURATION .

    ops InCoin:_ OutCoin:_ OutChoc:_ MchDelay:_
        CoinLight:_ ChocLight:_ ChocButton:_
        : Bool -> Attribute [ctor] .
    op Interface : -> Cid [ctor] .
endm
```

**Fig. 4.** Maude model of the interface

The automaton for the control states of the machine is given in Fig. 5. Changes of data attributes are annotated on the transitions in curly brackets. The machine initially is in the `Ready` state and is prepared to accept one coin. After a coin is inserted, the machine switches to `CoinAccepted` where it can delay some time before accepting the selection of a chocolate bar which triggers a transition to `ChocSelected`. When the change is given, the machine is in state `ChangeGiven` and either proceeds to `Ready`, or to `Stopping` if there is no more stock of chocolate bars or change coins.

Data attributes `Euro`, `50cent` and `Choc` hold as a value the number of 1 € coins and 50 cent coins and chocolate bars respectively. The Maude code defining these attributes is shown in Fig. 1. The actions to change the control state depend on the current state- and data-attributes as well as on the state of the interface:

**acceptcoin** There must be a 1 € coin in the `InCoin` slot as a precondition. The effect of this action is to move the € coin from the `InCoin` slot into the machine (Fig. 2).

**acceptcoin-not-ready** A coin was inserted while the machine was not expecting it. The coin is accumulated to the possessions of the machine, but no other action is triggered (the machine "swallows" the coin).

**delaying** After accepting the coin, the machine is ready to accept the chocolate selection. The machine needs some time to process the information communicated by the user (chocolate selection) and this causes a device-delay. We model this device-delay by setting the `MchDelay` attribute of the interface model to true.

**selection-accepted** This action releases the chocolate button that has been pressed by the user.

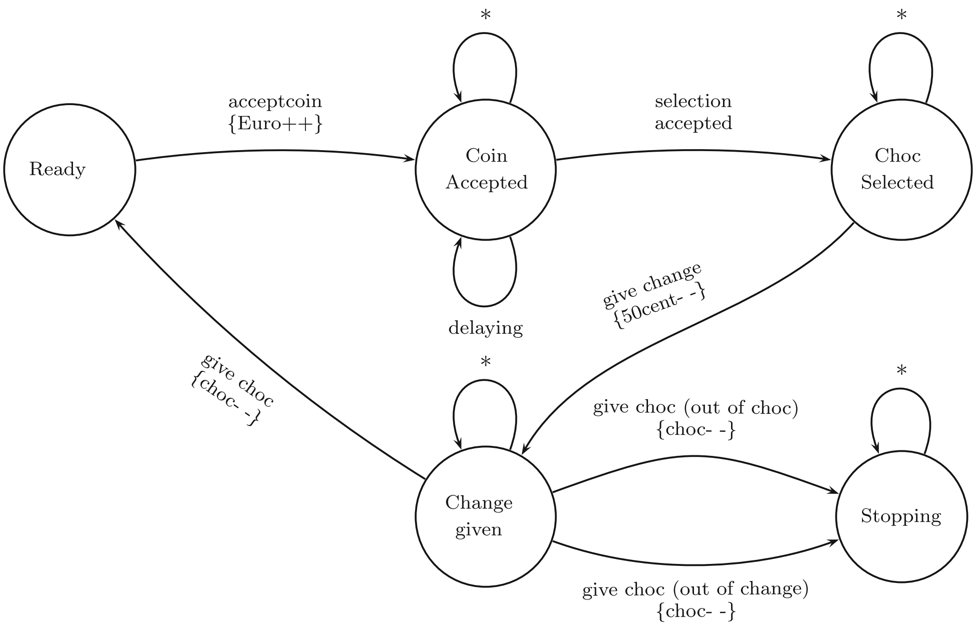

**Fig. 5.** Machine model (* represents the action *acceptcoin-not-ready*{*Euro*++})

**givechange** The machine gives the change and decreases the number of 50 cent coins by 1. A 50 cent coin is released to the `OutCoin` slot.

**givechoc** The machine is designed to give chocolate after giving the change coin. By doing this, the number of chocolate bars in the machine is decreased by 1 and a chocolate bar is released to the chocolate slot. The machine state changes to state `Ready` only if the machine still has chocolate bars and 50 cent coins in it. Otherwise it changes to the `Stopping` state.

The `givechoc` action is implemented as three rewrite rules. One rule is taken when the machine still has enough chocolate and 50 cent coin after giving chocolate. This rule changes the machine state to `Ready`. The other two rules are taken if there is no more chocolate or there are no more 50 cent coins in the machine after giving chocolate. In these cases the machine changes to the `Stopping` state.

Figure 2 shows an example of a Maude rule that represents the `accepting coin` action. In the left hand side of the rule we show the precondition in which this rule is applicable: there is a coin in the `InCoin` slot, the `CoinLight` is on, the `ChocLight` is off, and the machine is in the `Ready` state. On the right hand side we show the effect of the application of this rule: changing the machine state into `CoinAccepted`, increasing the number of € coins, turning off the `CoinLight` and turning on the `ChocLight`. The `ChocButton` is turned off to indicate that user actions other than `inserting coin` are ignored.

The rewrite rule defining the `delaying` action does not change the machine state, so the `delaying` action could also be defined in the `accepting coin` rule as a side effect. However we have chosen to define it separately for emphasis and for debugging purposes (the name of the rewrite rule is shown in the trace of Maude execution).

```
rl [ delaying ] :
   < int : Interface | CoinLight: false, ChocLight: true, MchDelay: false, A >
   < machine : Machine | State: CoinAccepted, A’ >
=>
   < int : Interface | CoinLight: false, ChocLight: true, MchDelay: true, A >
   < machine : Machine | State: CoinAccepted, A’ >.
```

```
rl [ selection-accepted ] :
   < int : Interface | ChocLight: true, InCoin: false, ChocButton: true, MchDelay: C, A >
   < machine : Machine | State: CoinAccepted, A' >
=>
   < int : Interface | ChocLight: false, InCoin: false, ChocButton: false,
                       MchDelay: false, A >
   < machine : Machine | State: ChocSelected, A' >.
```

We model non-determinism by allowing the machine not to take the `delaying` action. The rewrite rule for `selection-accepted` does not have a device-delay as a precondition (the expression "`C`" in "`MchDelay: C`" is a boolean variable). After accepting the coin, the machine may take the `delaying` action before `selection-accepted`, or may take `selection-accepted` directly. This non-determinism corresponds to the fact that time tolerance for each user may be different. Some users may notice this device-delay and some may not.

### 4.3. The user model

The model of the user maintains 4 data attributes: the number of 1 € and 50 cent coins is recorded in the `UEuro` and `U50cent` attributes. `UChoc` represents the number of chocolate bars the user has, and `WantChoc` indicates that the user wants more chocolate. The Maude representation of a user who wants a chocolate, has 5 Euros but has no chocolate bars looks like this:

```
< user : User | UState: Idle, UEuro: 5, U50cent: 0, UChoc: 0,
WantChoc: true >
```

The goal of the user interacting with a chocolate machine is to get a chocolate bar. The user knows that a 1 € coin has to be inserted and the chocolate has to be selected to achieve the goal. In general, the user does not know the order of actions to perform when interacting with the machine. We model this behaviour using the control states as follows:

**Idle** The user is not interacting with the machine
**Interacting** The user has just started the interacion and has not done any action
**CoinInserted** The user has inserted the 1 € coin but has not pressed the chocolate button
**ChocPressed** The user has pressed the button to select chocolate but has not inserted a coin
**BothDone** The user has inserted one coin and selected chocolate
**BothDone2** The user has redone the chocolate selection
**ChocTaken** The user has taken the chocolate bar from the machine
**ChangeTaken** The user has taken the 50 cent change coin from the machine

We model a user, who tries to achieve the goal by performing the obvious actions to insert the money and press the button, but can also choose to react to a machine output when the signal light indicates a required action. In detail, the user can perform the following actions:

**approaching the machine** The user who wishes to have chocolate and has money approaches the machine. After performing this action the user's control state is changed to `Interacting`.

**inserting coin** The user must have money to perform this action. The number of user's € coin is decreased by 1. If the user has not pressed the chocolate button, this action changes the control state to `CoinInserted`. If the user pressed the chocolate button before inserting the money, the user's mental state changes to `BothDone`.

**pressing chocolate button** The user may perform this action either before or after inserting a 1 € coin. The state is changed to `ChocPressed` or `BothDone`, depending on whether the 1 € coin was already inserted.

**getting chocolate bar** The user can take the chocolate bar any time after the machine has dispensed it to the chocolate slot.

**getting 50 cent coin** The user can take the 50 cent coin any time the machine has released it to the change coin slot.

**redoing an action** The only action a user will redo is pressing the chocolate button, since this action will not reduce user's possessions (1 € coin, 50 cent coin and chocolate bar). The user can redo chocolate selection only after observing device-delay and seeing the chocolate light still blinking.

**leaving the interaction** There are two reasons why the user may leave the interaction: goal achievement or impatience in waiting

**wanting chocolate** The user may want chocolate again after getting a chocolate bar and may approach the machine a second time.

In this case study there are two communication goals: inserting a coin and selecting the chocolate. These actions are the first two actions the user performs after approaching the machine. We model two kinds of user behaviours that trigger these actions: goal-based user and reactive user. Each user behaviour is modelled as a set of rules. The rules for goal-based behaviour consider the current state of the user as precondition for each action the user can take. On the other hand, the rules for reactive behaviour consider stimuli from the interface as precondition for each action the user can take. Combining both behaviours in one user model results in a complex user model. We show here the Maude code to model inserting coin action in our combined user model:

```
crl [ insertcoin ] :
    < int : Interface | InCoin: false, OutChoc: false, OutCoin: false, A >
    < user : User | UEuro: N, UState: Interacting, WantChoc: true, A' >
 =>
    < int : Interface | InCoin: true, OutChoc: false, OutCoin: false, A >
    < user : User | UEuro: sd(N, 1), UState: CoinInserted, WantChoc: true, A' >
 if N > 0.

crl [ insertcoin-after-presschoc ] :
    < int : Interface | InCoin: false, OutChoc: false, OutCoin: false, A >
    < user : User | UEuro: N, UState: ChocPressed, WantChoc: true, A' >
 => < int : Interface | InCoin: true, OutChoc: false, OutCoin: false, A >
    < user : User | UEuro: sd(N, 1), UState: BothDone, WantChoc: true, A' >
 if N > 0.

crl [ reactive-insertcoin ] :
    < int : Interface | CoinLight: true, InCoin: false, OutChoc: false,
                        OutCoin: false, A >
    < user : User | UEuro: N, UState: S, WantChoc: true, A' >
 =>
    < int : Interface | CoinLight: true, InCoin: true, OutChoc: false,
                        OutCoin: false, A >
    < user : User | UEuro: sd(N, 1), UState: CoinInserted, WantChoc: true, A' >
 if S =/= ChocPressed /\ S =/= CoinInserted /\ S =/= Idle /\ S =/= BothDone /\ N > 0.
```

The `insertcoin` and `insertcoin-after-presschoc` rewrite rules describe inserting a coin as a goal-based action, that can only be done when the user has not inserted any coins yet (`InCoin: false`). The OutCoin slot and OutChoc slot are included in the precondition to prevent the user from inserting another coin after a chocolate bar has been released by the machine. The `reactive-insertcoin` rewrite rule describes inserting coin as an action that will be taken by the user after the coin light is blinking. The additional conditions (checking the user state) is needed as a result of combining both behaviours. Note how object attributes irrelevant to the action are subsumed in the placeholders `A` and `A'`.

The redoing action is defined as a rewrite rule. This action can only be taken after the user has completed both communication goals and, during a device-delay can still see the chocolate light blinking.

```
rl [ reactive-presschoc-after-BothDone ] :
   < int : Interface | ChocLight: true, ChocButton: C, MchDelay: true, A >
   < user : User | UState: BothDone, WantChoc: true, A' >
 =>
   < int : Interface | ChocLight: true, ChocButton: true, MchDelay: true, A >
   < user : User | UState: BothDone2, WantChoc: true, A' >.
```

This rewrite rule changes the user state from `BothDone` to `BothDone2`. In this way the user can only redo an action once. Since we do not model real time, we must bound the redoing action, otherwise this action can be taken infinitely often and the machine cannot continue. If we model real time, we can define the duration of device-delay and we do not need to bound the number of redoing actions taken by the user.

The leaving action is implemented as three rewrite rules. Each rule is used to model a different reason for the user to leave the interaction. Under normal circumstances, a user would leave the interaction after achieving the intended goal. This is indicated by the value of attribute UState (`UState: ChocTaken`). Alternatively, a user may leave the interaction due to impatience during a device-delay. This is implemented by two more rewrite rules. As stated in Sect. 4.1, we assume that the user has a tolerance time while waiting for the result of the last action. However, in order to model this tolerance time explicitly we would need a tool that supports real-time modelling. In this paper we choose not to take that approach. We have used non-determinism to model user impatience. If a device delay occurs (indicated by the `MchDelay: true`) the user may perform one of these actions:

- leave the interaction,
- redo the chocolate selection action,
- do not perform any action and wait until the machine shows any result.

We assume that if a user chooses to leave the interaction upon device-delay, it means the delay is longer than the user's tolerance time. The following Maude code shows how we implemented the leaving action.

```
rl [ leaving-after-goal-achieved ] :
     < user : User | UState: ChocTaken, A >
  =>
     < user : User | UState: Idle, A >.

rl [ leaving-on-delay ] :
     < user : User | UState: BothDone, A >
     < int : Interface | MchDelay: true, Wait: false, A' >
   =>
     < user : User | UState: Idle, A >
     < int : Interface | MchDelay: true, Wait: false, A' >.

rl [ leaving-on-delay-after-retry ] :
     < user : User | UState: BothDone2, A >
     < int : Interface | MchDelay: true, Wait: false, A' >
   =>
     < user : User | UState: Idle, A >
     < int : Interface | MchDelay: true, Wait: false, A' >.
```

To put together machine, interface and user in an interaction, we define a configuration of these three objects and initialise their attributes. We then define properties that we are interested to check on the interaction and use the Maude LTL model-checker to check them. Figure 6 shows the Maude module that defines this interaction. HCI-TEST-PREDS-1 is the module that contains the definition of all the LTL predicates used for model- checking.

## 5. Checking errors

This section gives details on the result we got from model-checking errors occurring in the interaction between a user and a machine. Since Maude supports LTL model-checking, we define LTL properties and use Maude to verify them. The first property we are interested to verify specifies that the interaction is completed successfully. The interaction is successful when the user can achieve his goal during the interaction.

*Property 1* (*Success*): Every time a user starts an interaction, the intended goal will eventually be achieved. This property is defined in LTL as follows.

$\Box(\mathit{Approaching} \rightarrow \Diamond \mathit{GoalAchieved})$

Predicate `Approaching` indicates that the user has approached the chocolate machine. Implicitly it means that the user has a goal (wants chocolate) and has enough money. Predicate `GoalAchieved` indicates that the user has achieved the intended goal. We use the `WantChoc` attribute to check this predicate. We assume that before getting the chocolate the user will always want chocolate (`WantChoc` = true) and only after getting the chocolate the user does not want chocolate anymore.

```
in HCI-TEST-PREDS-1

mod HCI-TEST-CHECK-1 is
    pr HCI-TEST-PREDS-1 .
    inc MODEL-CHECKER .
    inc LTL-SIMPLIFIER .

   ops UA MCH I : -> Oid [ctor] .
   op init : -> Configuration .

   eq init = < UA : User | UState: Idle, UChoc: 0, U50cent: 0, UEuro: 3, WantChoc: true >
         < I : Interface | InCoin: false, OutCoin: false, OutChoc: false,
                 CoinLight: true, ChocLight: false, Wait: false,
                             MchDelay: false, ChocButton: false >
         < MCH : Machine | State: Ready, Choc: 5, 50cent: 8, Euro: 0 > .

endm
```

**Fig. 6.** Maude model of the interaction

The following Maude code specifies this predicate.

```
eq < user : User | WantChoc: false, A > C |= GoalAchieved = true .
eq C |= GoalAchieved = false [owise].
```

Property 1 only states that every approaching user will eventually achieve the intended goal, but does not say when the user achieves it. The user may approach the machine, leave temporarily and then return to be involved in a new interaction and finally achieve the intended goal. The next property is a stronger version of Property 1.

*Property 2* (*Success Before Leave*): Every time a user starts an interaction and the machine works properly, the user will always achieve the intended goal before leaving the interaction. This property is defined in LTL as follows.

$$\Box(\ (\mathit{Approaching}\ \wedge\ \mathit{Enabled}) \rightarrow (\sim \mathit{UserLeft}\ \ U\ \ \mathit{GoalAchieved})\ )$$

Predicate `Enabled` states that the machine can work properly. Predicate `Enabled` is true when the machine still has chocolate and 50 cent coins. There are cases in which the user approaches the machine but the machine is not ready (for instance there is no chocolate left in the machine). In theses cases it is impossible for the user to achieve the intended goal. To exclude these cases from Property 2, we add predicate `Enabled` in the formula above. Predicate `UserLeft` indicates that the user has left the interaction.

When we model-check this property, we get a counter-example that shows that the property is not satisfied by our model. It means that some errors may occur in the interaction. By analysing the counter-example, we can find out that the error is caused by the strict behaviour of the machine. When the order of actions that a user performs is different from the order of actions allowed by the machine, an *order error* occurs. To capture our observation of this error more explicitly, we define the following property.

*Property 3* (*Order Error*): If the user performs the chocolate selection as the first action, then only a redo (of the chocolate selection) allows the user to continue the interaction. This property is formally defined as follows.

$$\Box(\ (\sim \mathit{ChgGiven} \wedge\ (\mathit{Ready}\ \ U\ \ \mathit{ChocPressed}) \wedge\ \mathit{Ready}) \rightarrow (\sim \mathit{ChgGiven}\ \ U\ (\mathit{Redo} \vee\ \mathit{UserLeft}))\ )$$

In this property, `ChgGiven` is used to show that the machine accepts any order of actions, since the 50 cent coin is only released after the machine accepts all communication goals. Predicate `Ready` indicates that the machine is ready to accept user's first action, and `ChocPressed` is used to indicate that the user has pressed the chocolate button. Predicate `Redo` indicates that the user has redone chocolate selection action.

The model-checking analysis shows that this property is satisfied. Another property that is related with order error is flexibility of accepting user actions.

*Property 4* (*Flexibility*): If the machine works properly, then all communication goals given by the user will eventually be accepted. This is defined by the following LTL formula.

$$\Box((\mathit{Enabled} \wedge \mathit{BothDone}) \rightarrow\ \Diamond \mathit{ChgGiven})$$

Predicate `BothDone` states that the user believes all communication goals have been performed. Again we use `ChgGiven` to show machine's acceptance of user's communication goals.

This property not only shows freedom of order error, but also freedom of another kind of error. The user can perform both actions (inserting coin and selecting chocolate) before the machine is ready, which will violate property 4. This user behaviour may possibly be caused by *user habituation*. In our user model, the behaviour of a habituated user can be seen as a subset of a goal-based user. The goal-based user knows that in order to achieve the goal, two actions have to be performed. If the user is already habituated to a machine behaviour, these two actions can be performed without waiting any signal from the machine that indicates that the machine is ready. Model-checking this property on this model also gives a counter-example, meaning that both errors may occur.

Besides order error, there are two more errors that Basuki has successfully modelled in his previous work [Bas07], post-completion error and device-delay error. *Post-Completion error* occurs after the user achieves the goal, but forgets to complete a subtask. In our case study (i.e. after getting the chocolate bar), the user may forget taking 50 cent coin after achieving the goal. We define a property to check whether a machine is free of post-completion error.

*Property 5*: A user who achieves the intended goal will not leave the interaction before taking the change coin. We call this property *Post-Completion Error Free property*. It is formally defined as follows.

$$\Box((Enabled \wedge Approaching \wedge (\sim UserLeft\ U\ GoalAchieved)) \rightarrow (\sim UserLeft\ U\ ChgTaken))$$

Predicate `ChgTaken` indicates that the 50 cent coin has been taken by the user.

We now define a property to check whether device-delay may inhibit the user to achieve the intended goal.

*Property 6*: A user who starts an interaction and later experiences a device-delay, will eventually achieve the intended goal before leaving the interaction. We call this property *Device-Delay Error Free property*. It is formally defined as follows.

$$\Box((Enabled \wedge Approaching \wedge (\sim UserLeft\ U\ Delayed)) \rightarrow (\sim UserLeft\ U\ GoalAchieved))$$

Predicate `Delayed` indicates a device-delay is occurring in the machine. We model this by making use of the `MchDelay` attribute of the interface.

The model-checking analysis shows that Property 5 and 6 are not satisfied in this model. The counter-example for property 5 shows that although the machine releases the change coin before releasing the chocolate bar, the user may still take the chocolate bar and leave the interaction. In the case of property 6, the counter-example shows that upon device-delay, the user may leave the interaction immediately.

## 5.1. Chocolate machine with flexible ordering

We try to modify the machine design step by step. At each step we attempt to remove one kind of error. The first error we try to avoid is the order-error (detected when Property 3 is true). We design a new machine which allows flexible order of actions. In the initial state both lights blink to indicate that both actions (inserting coin and selecting chocolate) are enabled. After doing one action, the user must wait until the action is accepted and the machine is ready to accept another action. If the user performs the action before the machine becomes ready, the machine will ignore the action. In this case the user may either redo this action or leave the interaction.

To implement the flexibility in the order of actions, we add one more control state to the machine. Control state `BothAccepted` expresses that the machine has accepted both communication goals. This control state can be reached from `CoinAccepted` by accepting chocolate selection, or from `ChocSelected` by accepting coin insertion. This new control state differs from `BothDone` control state. The difference lies on whose point of view the control states represent. Control state `BothDone` represents user's belief of having given both communication goals, while control state `BothAccepted` represents machine's acceptance of these two communication goals.

The model-checking analysis shows that Property 3 is not satisfied by the model, indicating that the order-error no longer occurs using the modified machine. However, Property 4 is not satisfied indicating that every action performed by the user before the machine is ready will be ignored. The other properties are also not satisfied indicating that post-completion error and device-delay error still occur in the interaction.

## 5.2. Fully flexible chocolate machine model

This new design aims to provide full flexibility in user actions. In this machine, the user can perform all communication goals without waiting for the machine to be ready. The machine still processes the coin insertion before the chocolate selection but it does not ignore the chocolate selection action. By designing this machine we expect to avoid order error and enable habituated users to perform their actions correctly.

To implement the full flexibility, we only need to slightly modify the model by omitting the `ChocButton` attribute from the accepting-coin rewrite rule.

```
rl [ acceptcoin ] :
     < int : Interface | InCoin: true, CoinLight: true, ChocLight: false,
                         A >
     < machine : Machine | Euro: N, State: Ready, A' >
  =>
     < int : Interface | InCoin: false, CoinLight: false, ChocLight: true,
                         A >
     < machine : Machine | Euro: N + 1, State: CoinAccepted, A' >.
```

The model-checking analysis shows that an interaction between a user and this machine satisfies Flexibility property. However other kinds of error still occur in this model, as shown by the failure of model-checking Properties 1, 2, 5 and 6.

## 5.3. Avoiding post-completion error

The simplest way to avoid post-completion error is by forcing the user to take the 50 cent coin before taking the chocolate bar. We design a machine that has a mechanism to check whether the user has taken the 50 cent coin, and only gives the chocolate bar when the check succeeds. With this mechanism, we can force the user to take the 50 cent coin before taking the chocolate bar.

This mechanism is modelled by checking the `OutCoin` attribute of the interface. The rewrite rule for giving chocolate action has `OutCoin: false` as one of its preconditions.

```
crl [ givechoc ] :
     < int : Interface | OutChoc: C, InCoin: false, OutCoin: false,
                         CoinLight: C', A >
     < machine : Machine | Choc: N, 50cent: M, State: ChangeGiven, A' >
  =>
     < int : Interface | OutChoc: true, InCoin: false, OutCoin: false,
                         CoinLight: true, A >
     < machine : Machine | Choc: sd(N, 1), 50cent: M, State: Ready, A' >
     if N > 1 /\ M > 0.
```

The model-checking analysis shows that an interaction between a user and this machine satisfies the Flexibility and the Post-Completion Error Free properties. However Properties 1,2 and 6 are still not satisfied by this machine.

## 5.4. Avoiding device-delay error

To avoid device-delay error we must design a machine that prevents the user from leaving the interaction when device-delay occurs. We assume that the user leaves the interaction without achieving the intended goal believing that there is something wrong with the machine. We design a final machine that has a wait light that blinks to inform the user to wait for a moment when a device-delay occurs. We assume that when the wait light blinks, the machine needs some time to process information, and will eventually gives the change coin. Therefore, the user has to wait until the change coin is given. To model this, we must modify the interface. We add a new attribute `Wait`, that represents a waiting light that warns the user to wait. We remove the delaying rule and change it into a new rewrite rule that warns the user to wait.

```
rl [ informing-delay ] :
  < int : Interface | CoinLight: false, ChocLight: true, MchDelay: false, Wait: C', A >
=>
  < int : Interface | CoinLight: false, ChocLight: true, MchDelay: true, Wait: true, A >.
```

We also modify our user model, especially for the redo and leaving action. In the new model, these actions are taken only when the user observes device-delay and the waiting light is off.

```
rl [ reactive-presschoc-after-BothDone ] :
   < int : Interface | ChocLight: true, ChocButton: C, MchDelay: true, Wait: false, A >
   < user : User | UState: BothDone, WantChoc: true, A' >
=>
   < int : Interface | ChocLight: true, ChocButton: true, MchDelay: true,
                       Wait: false, A >
   < user : User | UState: BothDone2, WantChoc: true, A' >.

rl [ leaving-on-delay ] :
   < user : User | UState: BothDone, A >
   < int : Interface | MchDelay: true, Wait: false, A' >
=>
   < user : User | UState: Idle, A >
   < int : Interface | MchDelay: true, Wait: false, A' >.

rl [ leaving-on-delay-after-retry ] :
   < user : User | UState: BothDone2, A >
   < int : Interface | MchDelay: true, Wait: false, A' >
=>
   < user : User | UState: Idle, A >
   < int : Interface | MchDelay: true, Wait: false, A' >.
```

Interaction in the new model satisfies Flexibility, Post-Completion Error Free, Device-Delay Error Free, Success and Success Before Leave property.

## 6. Conclusion and future work

In his previous work [Bas07], Basuki used a process algebra based approach to model human–computer interaction. The chocolate machine case study from Curzon and Blandford [CB00b, CB00a, CB01] is used to reason about human-errors and to show how to avoid these errors by modifying the machine interface.

In this article, we have extended Basuki's work and overcome some of its limitations. We describe a framework to model human–computer interaction based on the interacting components model. Using this framework we can incorporate distinct user behavioural attitudes, within the same user model. This could not be achieved by Basuki's previous work, in which distinct user behavioural attitudes could not coexist within a single user model.

In Basuki's previous approach the design starts with a basic user model which is defined as a process, and is then extended by adding a set of constraints defined as additional processes, in order to model a more complex user. Each constraint process is composed in parallel with the basic user process through a set of synchronising events. The resultant process is then composed in parallel with the machine model. In this approach the behaviour of a complex user model is a subset of the behaviour of the basic user model. This implies that the basic model needs to be designed in such a way to incorporate all user attitudes that may potentially lead to errors. However, such attitudes are not all known while designing the basic model; some of them are actually identified only after interpreting the results of the model-checking analysis performed on an initial version of the basic model. A first problem with Basuki's approach is that the basic user model has to be redesigned every time such a situation occurs.

In the approach presented in this article, we can start by defining a simple user model that has only a few simple actions. Gradually we add new rules that introduces new actions and define a more complex behaviour. In fact, by introducing new actions in the user model, we enrich the behaviour rather than constraining it. In this way we can include new user's attitudes without any change to modules and rules already designed. The interacting components model used in our approach also offers decoupling of the user and machine models. The user and machine models are defined separately, and run asynchronously. The interface is the only medium of communication between the user and the machine.

A second problem of Basuki's previous approach is that distinct user behavioural attitudes could not coexist within a single user model. This is due to the fact that Basuki uses a goal-based user as a general user model and applies a constraint to it to define a reactive user. However, the addition of this constraint leads to the loss of the goal-based part of the user behaviour. This could have probably been solved in Basuki's approach by defining a more general basic user model which could support a refinement where both the goal-based and reactive behaviours could cohexist. However, the complexity needed to define such a general user model would make it too difficult to understand the model and, as a consequence, the approach. The coexistence of the two behaviours is not a problem in our rewrite system approach, since rules added to model one behaviour enrich the behaviour rather than reducing it.

We exploit the capability of our new approach by successfully analysing more properties than in Basuki's previous approach. The new framework also allows us to model a new kind of human error, which is caused by user habituation. This kind of error needs independent modelling of user actions and machine actions and could not be modelled in Basuki's approach, where user and machine are tightly coupled. In fact, our use of the interacting component model led to a more natural way to describe interactive systems than the approach used in Basuki's work. The use of an interface model that prevents direct communication between user and machine better reflects the way this kind of communication happens in reality.

As future work, we would like to explore the capability of our approach by modelling different case studies. We would also like to investigate the scalability of our approach by modelling a system that includes more than two components.

## Acknowledgments

The authors would like to thank Einar Broch Johnsen for helpful discussions and useful suggestions concerning the Maude rewriting system. A. Griesmayer and R. Schlatte are supported by the EU-project CREDO (IST-33826).

## References

[Bas07] Basuki TA (2007) Model-checking interface design to reduce user errors. In: Curzon P, Cerone A (eds) The pre-proceedings of the 2nd international workshop on formal methods for interactive systems (FMIS 2007), number RR-07-08 in Technical Report, pp 1–16. Queen Mary, University of London, September 2007. ISSN 1470-5559

[BBD00] Butterworth R, Blandford A, Duke DJ (2000) Demonstrating the cognitive plausability of interactive systems. Formal Aspects Comput 12:237–259

[BMPC98] Butler RW, Miller SP, Potts JN, Carreno VA (1998) A formal methods approach to the analysis of mode confusion. In: Proc. of the 17th Digital avionics systems conference. Washington, Oct 31–Nov 6, 1998

[CB00a] Curzon P, Blandford A (2000) Reasoning about order errors in interaction. In: Aagaard M, Harrison J, Schubert T (eds) The Supplementary Proceedings of the 13th international conference on theorem proving in higher order logics, pp 33–48, Portland U.S., August 2000. Oregon Graduate Institute, Oregon

[CB00b] Curzon P, Blandford A (2000) Using a verification system to reason about post-completion errors. In: Palanque PA, Paterno F (eds) Participants Proc. of DSV-IS 2000: 7th Int. workshop on design, specification and verification of interactive systems, at the 22nd Int. Conf. on Software Engineering, pp 292–308

[CB01] Curzon P, Blandford A (2001) Detecting multiple classes of user errors. In: Little MR, Nigay L (eds) Proceedings of the 8th IFIP international conference on engineering for human–computer interaction, pp 57–72

[CDE$^+$03] Clavel M, Durán F, Eker S, Lincoln P, Martí-Oliet N, Meseguer J, Talcott C (2003) The maude 2.0 system. In: Nieuwenhuis R (ed) Rewriting techniques and applications (RTA 2003). Lecture notes in computer science, vol 2706. Springer, Berlin, pp 76–87, June 2003

[CDE$^+$07] Clavel M, Durán F, Eker S, Lincoln P, Martí-Oliet N, Meseguer J, Talcott C (2007) Maude Manual (Version 2.3), July 2007

[CE06] Cerone A, Elbegbayan N (2006) Model-checking driven design of interactive systems. In: Cerone A, Curzon P (eds) Proceedings of the first international workshop on formal methods for interactive systems, pp 1–18

[CGP99] Clarke EM, Grumberg O, Peled DA (1999) Model checking. MIT Press, Cambridge

[Chi85] Chi UH (1985) Using model-checking to help discover mode confusions and other automation surprises. IEEE Trans Softw Eng SE-11(8):671–685

[CLS00] Claveland R, Li T, Sims S (2000) The Concurrency Workbench of the New Century User's Manual Version 1.2. SUNY at Stony Brook, Stony Brook, New York, June 2000. http://www.cs.sunysb.edu/cwb

[DCH97] Dwyer MB, Carr V, Hines L (1997) Model checking graphical user interface using abstractions. In: Software engineering ù ESEC/FSE'97. Lecture notes in computer science, vol 1301. Springer, Berlin, pp 244–261

[Dix91] Dix AJ (1991) Formal methods for interactive systems. Academic Press, New York

[DT04] Dwyer MB, Tkachuk O (2004) Analyzing interaction orderings with model checking. In: Proc. of ASE 2004, pp 154–163

[Hoa85] Hoare CAR (1985) Communicating sequential processes. International series in computer science. Prentice-Hall, Englewood Cliffs

[Joh97] Johnson C (1997) Reasoning about human error and system failure for accident analysis. In: Howard S, Hammond J, Lindgaard G (eds) Human–Computer Interaction INTERACT '97. Chapman and Hall, London, pp 331–338

[KP85] Kieras DE, Polson PG (1985) An approach to the formal analysis of user complexity. Int J Man–Mach Stud 22:365–394

[Lev97] Leveson NG et al (1997) Final report: a demonstration safety analysis of air traffic control software. NASA technical report, 1997. http://sunnyday.mit.edu/papers/dfw2.pdf

[LH03] Loer K, Harrison M (2003) Model-based formal analysis of temporal aspects in human–computer interaction. In: Proceedings of the HCI2003 workshop on the temporal aspects of tasks

[LNR87] Laird J, Newell A, Rosembloom P (1987) SOAR: an architecture for general intelligence. Artif Intell 33(1):164

[PBP97] Palanque PA, Bastide R, Paterno F (1997) Formal specification as a tool for objective assessment of safety-critical interactive systems. In: Howard S, Hammond J, Lindgaard G (eds) Human–computer interaction INTERACT '97. Chapman and Hall, London, pp 323–330

[Pnu77] Pnueli A (1977) The temporal logic of programs. In: IEEE symposium on foundations of computer science. Providence, RI, pp 46–57

[PS01] Paterno F, Santoro C (2001) Integrating model checking and HCI tools to help designers verify user interface properties. In: 7th international workshop, DSV-IS 2000 Limerick, Ireland. Lecture notes in computer science, vol 1946. Springer, Berlin, pp 135–150

[Rus02] Rushby J (2002) Using model-checking to help discover mode confusions and other automation surprises. Reliab Eng Syst Saf 75(2):167–177

[YGS89] Young RM, Green TRG, Simon T (1989) Programmable users models for predictive evaluation of interface design. In: ACM CHI 89 Human Factors in Computing Systems Conference. ACM Press, New York, pp 15–19

*Received 16 May 2008*

*Accepted in revised form 30 October 2008 by P. Curzon and D.A. Duce*

*Published online 8 January 2009*