UNIVERSIDAD POLITECNICA DE VALENCIA

DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN

Formal Modeling of Security Policies for Mobile Access Solutions

Master's Thesis

Presented by: Guillermo Roselló Gil

Supervisors: María Alpuente Frasnedo
Julia Sapiña Sanchis

Valencia – September 18th, 2017.

ABSTRACT

Current technology allows us to have a great deal of computation power in the palm of our hand in the form of smartphones. Such powerful and versatile devices are the perfect tools to, e.g., authenticate users in both virtual and physical systems. Key2phone is a smartphone solution designed to facilitate authorized users to access restricted spaces by turning their smartphones into electronic keys. This master's thesis focuses on the study of the safety and security properties of the Key2phone electronic lock. First, a model of the Key2phone protocol is implemented by using the Maude language, which is a high-performance reflective language and system supporting both equational and rewriting logic specification and programming. Then, a reachability analysis is performed in the Maude system, which ensures the safety of the system by proving that no states considered unsafe can be reached. Finally, liveness properties are checked by using the Maude LTL logical model checker (LMC).

RESUMEN

La tecnología actual nos permite tener un gran potencial de computación en la palma de la mano. Estos dispositivos, versátiles y potentes, son la herramienta perfecta para, por ejemplo, autorizar usuarios tanto en sistemas virtuales como físicos. Key2phone es una solución para *smartphone* diseñada para facilitar el acceso de usuarios autorizados a espacios restringidos, convirtiendo sus *smartphones* en llaves electrónicas. Esta tesis de master se centra en el estudio de las propiedades de seguridad del sistema Key2phone usando el lenguaje de programación Maude, que es un lenguaje reflectivo de alto rendimiento que soporta programación y especificaciones tanto de lógica ecuacional como de reescritura. Entonces, se realiza un análisis de alcanzabilidad, que comprueba la seguridad del sistema, probando que no se alcanzan estados del sistema considerados inseguros. Finalmente, se verifican propiedades de viveza y seguridad usando la herramienta *Maude LTL logical model checker (LMC).*

RESUM

La tecnologia actual ens permet tindre un gran potencial de computació en la palma de la mà. Estos dispositius, versàtils i potents, són la ferramenta perfecta per a, per exemple, autoritzar usuaris tant en sistemes virtuals com a físics. Key2phone és una solució per a *smartphone* dissenyada per a facilitar l'accés d'usuaris autoritzats a espais restringits, convertint els seus *smartphones* en claus electròniques. Esta tesi de màster se centra en l'estudi de les propietats de seguretat del sistema Key2phone usant el llenguatge de programació Maude, que és un llenguatge reflectiu d'alt rendiment que suporta programació i especificacions tant de lògica ecuacional com de reescriptura. Llavors, es realitza una anàlisi d'alcanzabilidad, que comprova la seguretat del sistema, provant que no s'aconseguixen estats del sistema considerats insegurs. Finalment, es verifiquen propietats de vivor i seguretat usant la ferramenta *Maude LTL logical model checker (LMC).*

CONTENTS

LIST OF FIGURES

CHAPTER 1 INTRODUCTION

We live in a world where communications are the keystone of our day to day. This is enhanced by the fact that almost everyone has a smartphone that takes part in his/her daily life in a very active way. Smartphones allow us to do almost everything, from checking and updating our social networks to using them for electronic payments by means of communication technologies like Bluetooth or Near Field Communications (NFC) [KC12].

Recently, the boundaries of what we can do with our smartphones have been pushed even further. We have the idea that our phone is kind of our personal identifier in the virtual world and we carry the smartphone every day at every moment. Then, what is keeping us from using our phone to identify us in the real world? A phone can be used, e.g., to unlock a hotel room, work office, house door; even to open garage and car doors. Although the use of a smartphone as an integrated door-key has several advantages, it also poses various safety and security vulnerabilities and threats. In the event that our smartphone is misplaced or stolen, we are totally vulnerable; also, how will the communication system behave during some severe disasters? For example, our workplace is on fire and people are trapped inside. Should the trapped people have to dial the door number and enter the Personal Identification Number (PIN) code to open the door? This can be a very time-consuming and unfeasible process, which may prove fatal when time is critical. Even the slightest vulnerability or threat in the system can severely hinder the safety and security of their owners, family, and goods. Hence, it is essential to provide a comprehensive analysis of personal safety and system security requirements. Further, there is a need for a suitable formal specification and verification model of the smartphone based door access control system in order to determine well behavior considerations and integrate them within the system's design. The resulting model could help the engineers to articulate what they must design in order to ensure the safety and security of the system. But a smartphone solution exists that wants to accomplish all this, the *Key2phone* system.

Key2phone [CLB$^+$15] is a mobile access solution for the management of electronic doors, turning one's smartphone into a key for electronic locks. Once the solution is deployed, the doors could be opened by typing the identifying number of each door or by reaching the system via Bluetooth connectivity. When the *door control module* detects a dialed number, it checks the user identification. In case the user belongs to an authenticated group, the module opens the door; if the user does not belong to any valid group, the call is simply ignored, which means that this request is free of computational load. In the case when the communication is done via Bluetooth, the door will only open when an authorized user, who has the Key2phone Bluetooth application running, reaches the Bluetooth range of the door. The access rights are managed online with a web-based configuration management system and access policies are transmitted to the Key2phone control module.

Safety and security are high-rank concerns in systems like Key2phone, where people physical safety is at stake. By physical safety we mean trying to avoid every scenario where a person may be injured. Also, we have to take care about the security of the system, with the promise of protection against any misuse by the authorized users.

Chaudhary et al. [CLB$^+$15] provide an study of the security threats/vulnerabilities and their manage control in an environment where Key2phone could be deployed. They identified different actors, preconditions, and assumptions for each possible scenario. The first phase of the study consisted in asking a group of experts in security and usability to make a list of the safety requirements for each scenario. In the second phase, external field experts participated to identify if there were any missing or unnecessary requirements.

The goal of this master's thesis is to study the trustworthiness of the Key2phone mobile access solution. Our analysis is based on using the high performance language and system Maude [CDE$^+$16], which effectively supports reachability analysis and model checking. A preliminary formal specification for the system using the Finite State Process (FSP) formal specification method, which is based on process algebra notation, and finite *Labeled Transition Systems* (LTS), was first given in [CLB$^+$15].

This manuscript is organized as follows. In Chapter 2, we provide an introduction to rewriting logic, the Maude language, and the Maude LTL logical model checker. In Chapter 3 we summarize the list of security threats and vulnerabilities discussed in [CLB$^+$15] and the resulting design of the Key2phone system. Chapter 4 describes the formal specification developed in this work and Chapter 5 analyses the properties to be verified. Finally, Chapter 6 provides some conclusions and discusses future work.

CHAPTER 2 PRELIMINARIES

In order to better understand this work, we recall the standard notions and terminology of term rewriting. We assume some basic knowledge of term rewriting [TeR03] and rewriting logic [Mes92]. Some familiarity with the Maude language is also assumed [CDE$^+$16].

This chapter is organized as follows. Section 2.1 addresses the notions of the rewriting logic. In Section 2.2 we introduce the fundamentals of the Maude language. Finally, in Section 2.3 the Maude LTL logical model checker capabilities are explained.

2.1 Rewriting logic

We assume an *order-sorted signature* $\Sigma = (S, \leq, \Sigma)$ with poset of sorts $(S, \leq)$, and an finite number of function symbols Σ. We also assume an S-sorted family $\mathcal{X} = \{\mathcal{X}_s\}_{s \in S}$ of disjoint variable sets with each $\mathcal{X}_s$ countably infinite. $\tau(\Sigma, \mathcal{X})_s$ and $\tau(\Sigma)_s$ are the sets of terms and ground terms of sort s, respectively. $\tau(\Sigma, \mathcal{X})$ and $\tau(\Sigma)$ for the corresponding term algebras. The set of variables that occur in a term t is denoted by $\mathcal{V}ar(t)$. In order to simplify the presentation, we often disregard sorts when no confusion can arise.

A *position* w in a term t is represented by sequence of natural numbers that addresses a subterm of t (Λ denotes the empty sequence, i.e., the root position). The set of positions of a term t is written $Pos(t)$, and the set of non-variable positions $Pos_{\Sigma}(t)$. The subterm of t at position w is $t|_w$, and

$t[u]_w$ is the result of replacing $t|_w$ by u in t.

A *substitution* σ is a mapping from variables to terms $\{X_1 \to t_1, \ldots, X_n \to t_n\}$ such that $X_i\sigma = t_i$ for $i = 1, \ldots, n$ (with $X_i \neq X_j$), and $X\sigma = X$ for all other variables X. Given a substitution $\sigma = \{X_1 \to t_1, \ldots, X_n \to t_n\}$, the *domain* of σ is the set $Dom(\sigma) = \{X_1, \ldots, X_n\}$. For any substitution σ and set of variables V, $\sigma_{\lceil V}$ denotes the substitution obtained from σ by restricting its domain to V (i.e., $X\sigma_{\lceil V}$ if $X \in V$, otherwise $X\sigma_{\lceil V} = X$). Given two terms s and t, a substitution σ is a *matcher* of t in s, if $s\sigma = t$. By $match_s(t)$, we denote the function that returns a matcher of t in s if such a matcher exists, otherwise $match_s(t)$ returns $fail$.

A *conditional* rule is an expression of the form $\lambda \to \rho$ *if* C, where $\lambda, \sigma \in \tau(\Sigma, \mathcal{X})$, and C is a (possibly empty) sequence $c_1 \wedge \ldots \wedge c_n$, where each c_i is an equational condition, a matching condition, or a rewrite expression. When the condition C is empty, we simply write $\lambda \to \rho$. A conditional rule $\lambda \to \rho$ *if* $c_1 \wedge \ldots \wedge c_n$ is *admissible* iff it fulfils the exact analogous of the admissibility constraints (i) and (ii) for the equational conditions and the matching conditions, plus the following additional constraint: for each rewrite expression c_i in C of the form $e \Rightarrow p$, $\mathcal{V}ar(e) \subseteq \mathcal{V}ar(\lambda) \cup \bigcup_{j=1}^{i-1} \mathcal{V}ar(c_j)$.

A Σ-*equation* is an unoriented pair $t = t'$, where $t, t' \in \tau(\Sigma, \mathcal{X})_s$ for some sort $s \in S$. Given Σ and a set E of Σ-equations, order-sorted equational logic induces a congruence relation $=_E$ on terms $t, t' \in \tau(\Sigma, \mathcal{X})$. The E-equivalence class of a term t is denoted by $[t]_E$, and $\tau(\Sigma/E, \mathcal{X})$ and $\tau(\Sigma/E)$ denote the corresponding order-sorted term algebras modulo E. An *equational theory* (Σ, E) is a pair with Σ and order-sorted signature and E a set of Σ a set of Σ-equations.

A *rewrite rule* is an oriented pair $l \to r$, where $l \notin \mathcal{X}$ and $l, r \in \tau(\Sigma, \mathcal{X})_s$ for some sort $s \in S$. An *(unconditional) order-sorted rewrite theory* is a triple $\mathcal{R}^c = (\Sigma, E, R)$ with Σ an order-sorted signature, E a set of Σ-equations, and R a set of rewrite rules. A *topmost rewrite theory* (Σ, E, R) is a rewrite theory s.t. for each $l \to r \in R, l, r \in \tau(\Sigma, \mathcal{X})_{State}$ for a top sort **State**, $r \notin \mathcal{X}$, and no operator in Σ has **State** as an argument sort. The rewriting relation $\to_R$ on $\tau_\Sigma(\mathcal{X})$ is $t \overset{p}{\to}_R t'$ (or $\to_R$) if $p \in Pos_\Sigma(t)$, $l \to r \in R, t|_p = l\sigma$, and $t' = t[r\sigma]_p$ for some σ. The relation $\to_{R/E}$ on $\tau_\Sigma(\mathcal{X})$ is $=_E; \to_R; =_E$, i.e., $t \to_{R/E} \iff \exists u_1, u_2 \in \tau_\Sigma(\mathcal{X})$ s.t. $t =_E u_1, u_1 \to_R u_2$, and $u_2 =_E s$. Note that $\to_{R/E}$ on $\tau_\Sigma(\mathcal{X})$ induces a relation $\to_{R/E}$ on $\tau_{\Sigma/E}(\mathcal{X})$ by $[t]_E \to_{R/E} [t']_E$ iff $t \to_{R/E} t'$. The transitive (resp. transitive and reflexive) closure of $\to_{R/E}$ is denoted $\to^+_{R/E}$(resp. $\to^*_{R/E}$). A term t is called $\to_{R/E}$-irreducible (or just

R/E-irreducible) if there is no term t' such that $t \rightarrow_{R/E} t'$. For a rewrite rule $l \rightarrow r$, we say that it is *sort-decreasing* if for each substitution σ, we have r$\sigma \in \tau_\Sigma(\mathcal{X})_s$. A rewrite theory (Σ, E, R) is sort-decreasing if all rules in R are. For a Σ-equation $t = t'$, we say that it is *regular* if $\mathcal{V}ar(t')$, and it is *sort-preserving* if for each substitution σ, we have $t\sigma \in \tau_\Sigma(\mathcal{X})_s$ implies $t'\sigma \in \tau_\Sigma(\mathcal{X})_s$ and vice versa. For substitution σ, ρ and a set of variables V we define $\sigma|_V \rightarrow_{R/E} \rho|_V$ if there is $x \in V$ such that $x\sigma \rightarrow_{R/E} x\rho$, and for all other $y \in V$ we have $y\sigma =_E y\rho$. A *substitution* is called R/E*-normalized* (or normalized) if $x\sigma$ is R/E-irreducible for all $x \in V$.

The relation $\rightarrow_{R/E}$ is called *terminating* if there is no infinite sequence $t_1 \rightarrow_{R/E} t_2 \rightarrow_{R/E} \ldots t_n \rightarrow_{R/E} t_{n+1} \ldots$. Further, the relation $\rightarrow_{R/E}$ is *confluent* if whenever $t \rightarrow^*_{R/E} t'$ and $t \rightarrow^*_{R/E} t^{''}$, there exists a term t''' such that $t' \rightarrow^*_{R/E} t'''$ and $t'' \rightarrow^*_{R/E} t'''$. An order-sorted rewrite theory (Σ, E, R) is confluent (resp. terminating) if the relation $\rightarrow_{R/E}$ is confluent (resp.terminating). In a confluent, terminating, sort-decreasing, order-sorted rewrite theory, for each term $t \in \tau_\Sigma(\mathcal{X})$, there is a unique (up to E-equivalence) R/E-irreducible term t' obtained from t by rewriting to canonical form, which is denoted by $t \rightarrow^!_{R/E} t'$, or $t \downarrow_{R/E}$ when t' is not relevant.

The relation $\rightarrow_{R/E}$-reducibility is undecidable in general since E-congruence classes can be arbitrarily large. Therefore, R/E-rewriting is usually implemented by R, E-rewriting, thanks to the notion of coherence. A relation $\rightarrow_{R,E}$ on $\tau_\Sigma(\mathcal{X})$ is defined as: $t \rightarrow_{p,R,E} t'$ (or just $t \rightarrow_{R,E} t'$) iff there is a non-variable position $p \in Pos_\Sigma(t)$, a rule $l \rightarrow r$ in R, and a substitution σ such that $t|_p =_E l\sigma$ and $t' = t[r\sigma]_p$. Note that, assuming E-matching is decidable, $\rightarrow_{R,E}$ is decidable. Notions such as confluence, termination, irreducible terms, and normalized substitution, are defined in a straightforward manner.

2.2 Maude

Maude is a very efficient implementation of rewriting logic [Mes92], which is publicly available at `http://maude.cs.illinois.edu`. Maude is a high-performance reflective language and system supporting both equational and rewriting logic specification and programming for a wide range of applications [CDE+16]. A Maude program is made up of different *modules*. Each module can include:

- *sort* (or type) declarations;

- *variable* declarations;
- *operator* declarations;
- *rules* and/or *equations* describing the behavior of the system operations, i.e., the *functions*.

Maude mainly distinguishes two kinds of modules depending on the constructions they define and on their expected behavior. Functional modules do not contain rules and their equations are expected to be confluent and terminating. On the contrary, system modules can contain both equations and rules, but their equations are also expected to be confluent and terminating though the behavior of its rules may be non-confluent and non-terminating. A functional module is enclosed by the reserved keywords `fmod` and `endfm`, whereas a system module enclosed in between `mod` and `endm` [CDE$^+$16].

2.2.1 Sorts

Maude *identifiers* are sequences of ASCII characters without white spaces, nor the special characters '[','], '(', ')', '{', '}', unless they are escaped with the back-quote character '`'. A sort is the type of the data being defined and they are declared as follows:

```
sort S .
```

where `S` is the identifier of the newly introduced sort `S`. If we want to introduce many sorts `S1 S2 ...Tn` at the same time, we write:

```
sorts S1 S2 ... Sn .
```

Sorts can be organized into *hierarchies* with `subsort` declarations. In the following declaration:

```
subsort S1 < S2 .
```

we state that each term of sort S_1 is also of sort S_2. For example, we can define natural numbers by considering their classification as positive numbers or as the zero number in this way:

```
sorts Nat Zero NonZeroNat .
subsort Zero < Nat .
subsort NonZeroNat < Nat .
op 0 :  -> Zero [ctor] .
op s :  -> Nat -> NonZeroNat [ctor] .
```

Maude also provides operator *overloading*. For example, if we add:

```
sort Binary .
op 0 :  -> Binary [ctor] .
```

to the previous declarations, the operator `0` is used to construct values both for the `Nat` and for the `Binary` sorts.

2.2.2 Operators

After having declared the sorts, we define operators. Operators are structures with a list of sorts as its arguments and a sort as a result. The scheme of an operator has the form:

```
op C : S1 S2 ... Sn -> S .
```

where `S1 S2 ... Sn` are the sorts of the arguments for operator `C`, and `S` is the resulting sort for the operator. We can also declare at the same time many operators `C1 C2 ... Cm` with the same signature (i.e., sort of arguments and resulting sort):

```
ops C1 C2 ... Cm :  S1 S2 ... Sn -> S .
```

Operators can be split into *constructors* and *defined* symbols. Constructor terms are made of constructor symbols and variables, and they constitute the *ground terms* or *data* associated to a sort, whereas defined symbols are used to construct calls to functions whose behavior will be specified by means of equations or rules. The rewriting engine of Maude does not distinguish between constructors or defined symbols, so there is no real syntactic difference between them. However, for documentation (and debugging) purposes, operators that are used as constructors can be labeled with the attribute `ctor`.

Operator attributes

Operator attributes are labels that can be associated to an operator in order to provide additional information (either syntactic or semantic) about the operator. All such attributes are declared within a single pair of enclosing square brackets "`[`" and "`]`":

$$C_1\ C_2 \ldots C_m : S_1\ S_2 \ldots S_n \to S[A_1 \ldots A_k]$$

where the A_i are attribute identifiers. The set of operator attributes includes among others: `ctor, assoc, comm, id, ditto,` etc.

Mix-fix notation

Another interesting feature of operators in Maude is *mix-fix* notation. Every operator defined as above is declared in *prefix* notation, that is, its arguments are separated by commas, and enclosed in parenthesis, following the operator symbol, as in:

$$C(s_1, s_2, \ldots, s_n)$$

where C is an operator symbol, and $s_1, s_2, \ldots, s_n$ are, respectively, terms of sorts $S_1, S_2, \ldots, S_n$. Nevertheless, Maude provides a powerful and tunable syntax analyzer that allows us to declare operators that are composed of different identifiers separated by its arguments. Mix-fix operators are identified by the sequence of its component identifiers with characters '`_`' inserted in the place each argument is expected to be, as follows:

```
op if_then_else_fi :  Bool Exp Exp -> Exp .

op __ :  Element List -> List .
```

The first line above defines an if-then-else operator, while the second one defines lists of juxtaposed (i.e., separated by white spaces) elements. A term built with the `if_then_else_fi` operator looks like:

```
if b1 then e1 else e2 fi
```

where the tokens `if`, `then`, `else`, and `fi` represent the mix-fix operator, `b1` represents a term of sort `Bool`, and finally `e1` and `e2` represent terms of sort `Exp`. A term built with the operator _ _ looks like:

```
e1 e2
```

where `e1` is a term of sort `Element`, `e2` is a term of sort `List`, and the space separating them represents the juxtaposition operator __ .

2.2.3 Structural axioms

The *Maude* language allows the specification of structural axioms over operators, i.e., certain algebraic properties like *associativity*, *commutativity*, and *unity* that operators may satisfy. In the following, we write A, C, U respectively to refer to these three algebraic properties. Structural axioms perform the computation on equivalence classes of expressions, instead of on simple expressions. In order to carry out computations on equivalence classes, Maude chooses an irreducible representative of each class and uses it for the computation. Thanks to the structural information, given as operator attributes, Maude can also choose specific data structures for an efficient low-level representation of expressions. For example, let us define a list of natural numbers separated by colons:

```
Sorts NatList EmptyNatlist NonEmptyNatList .
subsort EmptyNatList < NatList .
subsort Nat < NonEmptyList < NatList .
op nil :  -> EmptyNatList [ctor] .
op _:_ :  NatList Natlist -> NonEmptyNatList [assoc] .
```

The operator `_:_` is declared as associative by means of its attribute `assoc`. Associativity means that the value of an expression is not dependent on the subexpresion grouping considered, that is, the places where the parenthesis are inserted. Thus, if `_:_` is associative Maude considers the following expressions as equivalent:

```
s(0) : s(s(0)) : nil
(s(0) : s(s(0))) : nil
s(0) : (s(s(0)) : nil)
```

As another example, let us define an asociative list with `nil` as its identity element:

```
sort NatList .
subsort Nat < NatList .
op nil :  -> NatList [ctor] .
op _;_ :  NatList NatList -> NatList [assoc id: nill] .
```

The operator `_;_` is declared as having `nil` as its *identity element* by means of its attribute `id: nil`. Having an identity element e means that the value of an expression is not dependent on the presence of e's as subexpresions, that is, it is possible to insert e's without changing the meaning of the expression. Thus, in our example Maude considers the following expressions (and an infinite number of similar ones) as equivalent:

```
s(0) ;  s(s(0))
nil ; s(0) ; s(s(0))
s(0) ; nil ; s(s(0))
s(0) ; s(s(0)) ; nil
nil ; s(0) ; nil ; s(s(0)) : nil
⋮
```

For that reason, Maude omits nil in the irreducible representative, unless it appears alone as an expression. Now, let us introduce how we define a multiset, that is, an associative an commutative list with `nil` as its identity element:

```
sort NatMultiSet .
subsort Nat < NatMultiSet .
op nil :  -> NatMultiSet .
op _:_ :  NatMultiSet NatMultiSet -> NatMultiSet [assoc comm
id: nil] .
```

In this example, the operator `_:_` is declared to be *commutative* by means of the attribute `comm`. Commutativity means that the value of an expression is not dependent on the order of its subexpressions, that is, it is possible to change the order od subexpressions without changing the meaning of the expression. Thus, if `_:_` is a commutative and associative operator, Maude considers the following expressions equivalent:

```
s(0) :  s(s(0)) :  s(s(0))
s(s(0)) :  s(0) :  s(s(0))
s(s(0)) :  s(s(0)) :  s(0)
```

The structural properties are efficiently built in Maude. Additional structural properties can be defined by means of equations, as we discuss below.

Rules and equations

In Maude, *rules* and/or *equations* characterize the behavior of the *defined symbols*. Both language constructions have a similar structure:

```
eq l = r .
rl l => r .
```

where `l` and `r` are `terms`, i.e., expressions recursively built by nesting correctly typed operators and variables. `l` is called the left-hand side of a rule or equation, whereas `r` is its right-hand side. Both rules and equations can also be conditional. The behavior is similar, with the difference that for a conditional rule or equation to apply, there is a need to meet some conditions listed in the declaration of the rule or equation. In Maude, the declaration of conditional rules and equations is as follows:

```
ceq l = r if Condition-1 /\ ... /\ Condition-k .
crl l => r if Condition-1 /\ ... /\ Condition-k .
```

Variables

Variables can be declared when they are used in an expression by using the structure *name:sort*, or also in general variable equation:

```
var N1 N2 ... Nm :  S .
```

where `N1 N2 ... Nm` are variable names and `S` is a sort.

2.2.4 The `search` command

The `search` command allows one to explore (following a breadth-first strategy) the reachable state space in different ways. Throughout this thesis, we will use the `search` command this way:

```
search <term1> =>* <term2> .
```

where, `<term1>` is the starting term that will be used to explore the state tree, `<term2>` is the pattern that has to be reached, and `=>*` indicates that the rewriting proof from `<term1>` to `<term2>` can consist of zero or more steps.

2.3 Model checking in Maude

The Maude LTL model checker [SS03] supports on-the-fly explicit-state model checking of concurrent systems that are expressed as rewrite theories.

In every Maude system module, we can find two different levels of specification [CDE$^+$16]:

- a *system specification* level, provided by the rewrite theory specified by that system module which defines the behavior of the system.
- a *property specification* level, given by some property (or properties).

On the one hand, in order to check the *system specification* we only need to execute the specification in a Maude environment and see if its behavior is as desired. On the other hand, we need a specific property specification logic, e.g., *linear temporal logic* (LTL), and a decision procedure for it, e.g., *model checking*, to prove properties when the set of states that are reachable from an initial state in a system module is finite. To accomplish this, Maude uses all the modules specified in the file `model-checker.maude`.

Temporal logic allows one to specify properties such as safety properties, which ensure that something bad never happens; and liveness properties, which ensure that something good can eventually happen. These properties are related to the *infinite behavior* of a system. There are different temporal logics, but we focus on the linear temporal logic LTL, because of its intuitive appeal, widespread use, and well-developed proof methods and decision procedures.

2.3.1 LTL formulas and the `LTL` module

Given a set AP of *atomic propositions*, we inductively define the formulas of the *propositional linear temporal logic* over AP, LTL(AP), or simply LTL when no confusion can arise, as follows [CDE+16]:

- **True**: $\top \in$ LTL.
- **Atomic propositions**: If $p \in AP$, then $p \in$ LTL.
- **Next operator**: If $\varphi \in$ LTL, then $\circ\varphi \in$ LTL.
- **Until operator**: if $\varphi, \psi \in$ LTL, then $\varphi\ \mathcal{U}\ \psi \in$ LTL.
- **Boolean connectives**: if $\varphi, \psi \in$ LTL, then the formulas $\neg\varphi$ and $\varphi \vee \psi$ are in LTL.

Other LTL connectives can be defined in terms of the above minimal set of connective as follows:

- Other Boolean connectives:
  - **False**: $\bot = \neg\top$.
  - **Conjunction**: $\varphi \wedge \psi = \neg((\neg\varphi) \vee (\neg\psi))$
  - **Implication**: $\varphi \rightarrow \psi = (\neg\varphi) \vee \psi$.
- Other temporal operators:
  - **Eventually**: $\Diamond\varphi = \top\mathcal{U}\varphi$
  - **Henceforth**: $\Box\varphi = \neg\Diamond\neg\varphi$
  - **Release**: $\varphi\mathcal{R}\psi = \neg((\neg\varphi)\mathcal{U}(\neg\psi))$
  - **Unless**: $\varphi\mathcal{W}\psi = (\varphi\mathcal{U}\psi) \vee (\Box\varphi)$
  - **Leads-to**: $\varphi \leadsto \psi = \Box(\varphi \rightarrow (\Diamond\psi))$
  - **Strong implication**: $\varphi \Rightarrow \psi = \Box(\varphi \rightarrow \psi)$
  - **Strong equivalence**: $\varphi \Leftrightarrow \psi = \Box(\varphi \leftrightarrow \psi)$

The LTL syntax is a module that is included into the `MODEL-CHECKER` module. In Appendix B. Properties specification we provide the Maude code that defines this module.

2.3.2 Associating Kripke structures to rewrite theories

A Kripke structure is a total *unlabeled transition system* to which we add a collection of unary state predicates on its set of states.

Formally, a *Kripke structure* is a triple $\mathcal{A} = (A, \rightarrow_{\mathcal{A}}, L)$ such that A is a set, called the set of *states*, $\rightarrow_{\mathcal{A}}$ is a total binary relation on A, called the *transition relation*, and $L : A \rightarrow \mathcal{P}(AP)$ is a function, called the *labelling function*, that associates to each state $a \in A$ the set $L(a)$ of those *atomic propositions* in AP that *hold* in the state a [CDE+16].

Since the models of temporal logic are Kripke structures, we need to explain how we can associate a Kripke structure to the rewrite theory specified by a Maude system module [CDE+16].

In order to define the semantics of LTL, the following satisfaction relation is used:

$$\mathcal{A}, a \models \varphi$$

where $\mathcal{A}$ is the Kripke structure having AP as its atomic propositions, a is a state $a \in \mathcal{A}$ and, finally, the LTL formula $\varphi \in \mathrm{LTL}(AP)$. We say $\mathcal{A}, a| = \varphi$ if and only if, for each path $\pi \in Path(\mathcal{A})_a$, the *path satisfaction relation*

$$\mathcal{A}, \pi \models \varphi$$

holds, where we define the set $Path(\mathcal{A})_a$ of *computation paths* stemming from state a as a set of functions of the form $\pi : \mathbb{N} \rightarrow A$ such that $\pi(0) = a$ and, for each $n \in \mathbb{N}$ we have $\pi(n) \rightarrow_{\mathcal{A}} \pi(n+1)$.

These path satisfaction relation can be defined for any path, beginning at any state, inductively as follows:

- We always have $\mathcal{A}, \pi \models_{LTL} \top$.
- For $p \in AP$,

$$\mathcal{A}, \pi \models_{LTL} p \Leftrightarrow p \in L(\pi(0))$$

- For $\circ\varphi \in LTL(AP)$,

$$\mathcal{A}, \pi \models_{LTL} \varphi \mathcal{U} \psi \Leftrightarrow (\exists n \in \mathbb{N})$$
$$((\mathcal{A}, s^n; \pi \models_{LTL} \psi) \wedge ((\forall m \in \mathbb{N}) m < n \Rightarrow \mathcal{A}, s^m; \pi \models_{LTL} \varphi))$$

- For $\neg\varphi \in \mathrm{LTL}(AP)$,

$$\mathcal{A}, \pi \models_{LTL} \neg\varphi \Leftrightarrow \mathcal{A}, \pi \not\models_{LTL} \varphi$$

- For $\varphi \vee \psi \in \mathrm{LTL}(AP)$,

$$\mathcal{A}, \pi \models_{LTL} \varphi \vee \psi \Leftrightarrow \mathcal{A}\pi \models_{LTL} \varphi \text{ or } \mathcal{A}, \pi \models_{LTL} \psi$$

To accomplish the goal of associating the Kripke structure to the rewrite theory $\mathcal{R} = (\Sigma, E, \phi, R)$ specified by a Maude system module `M`, we need to make explicit two things:

- the intended *sort* k of states in the signature Σ, and
- the relevant *state predicates*, that is, the relevant set AP of atomic propositions.

These state predicates are just certain *predicates* about the state of the system specified by `M` that are needed to specify some system *properties* [CDE+16]. For this reason, these state predicates do not need to be part of the *system specification*; instead, they will be defined in the *property specification*. In order to clarify this, we going to take the system specification `M` and choose a given sort, say `Foo`, as our subsort of states. Once we choose this, we are able to specify the relevant state predicates in a different module called `M-PREDS` that includes `M`. This `M-PREDS` module is created following the pattern:

```
mod M-PREDS is
   protecting M .
   including SATISFACTION .
   subsort Foo < State .
   ...
endm
```

where the dots '. . . ' indicate the part in which the syntax and semantics of the relevant state predicates are specified, as further explained in what follows.

The module `SATISFACTION` is contained inside the `model-checker.maude` file, is very simple, and has the following specification:

```
 fmod SATISFACTION is
   protecting BOOL .
   sorts State Prop .
   op _|=_ :  State Prop -> Bool [frozen] .
endm
```

In this module, we can see that the two sorts `State` and `Prop` are left unspecified. This declaration will be done when we import the module `SATISFACTION` into our predicate module, using the following declaration:

```
   subsort Foo < State .
```

Now, all terms of sort `Foo` in `M` are also terms of sort `State`.

On the one hand, we have the `State` sort, which is related to the chosen sort in the *system module*; and on the other hand, we have the state predicates, which are defined in our *predicate module* by using appropriate equations. These state predicates are declared as operators of sort `Prop`, and now is when the operator

```
   op _|=_ :  State Prop -> Bool [frozen] .
```

defines the satisfiability of the given predicate by using suitable equations.

In order to illustrate how we can define the `M-PREDS` module, we are going to show an example, extracted from the chapter 10 of [CDE+16].

Example

Consider the following module `MUTEX` in which two processes, one named a and another named b, can be either waiting or in their critical section, and take turns accessing their critical section by passing each other a different `token` (either $ or *).

```
mod MUTEX is
 sorts Name Mode Proc Token Conf .
 subsorts Token Proc < Conf .
 op none : -> Conf [ctor] .
 op __: Conf Conf -> Conf [ctor assoc comm id: none] .
 ops a b : -> Name [ctor] .
 ops wait critical : -> Mode [ctor] .
 op [_,_] : Name Mode -> Proc [ctor] .
 ops * $ : -> Token [ctor] .
 rl [a-enter] : $ [a, wait] => [a, critical] .
 rl [b-enter] : * [b, wait] => [b, critical] .
 rl [a-exit] : [a, critical] => [a, wait] * .
 rl [b-exit] : [b, critical] => [b, wait] $ .
endm
```

For this example, we choose the sort `Conf` of configurations as our `State` sort. Now we want to check the liveness and safety of the MUTEX algorithm, and for this, we need state predicates that tell us whether a process is waiting or is in his critical section.

```
mod MUTEX-PREDS is
 protecting MUTEX .
 including SATISFACTION .
 subsort Conf < State .
 op crit : Name -> Prop .
 op wait : Name -> Prop .
 var N : Name .
 var C : Conf .
 var P : Prop .
 eq [N, critical] C |= crit(N) = true .
 eq [N, wait] C |= wait(N) = true .
 eq C |= P = false [owise] .
endm
```

In the above example we can see how one can specify the desired states by choosing a sort in `MUTEX` and declaring it as a subsort of `State`. Then, desired state predicates are defined as operators of sort `Prop`, by defining their semantics using a set of equations that specify for what states a given state predicate evaluates to true.

2.3.3 LTL model checking

Once we have a system module `M` that specifies a rewrite theory $\mathcal{R} = (\Sigma, E, \phi, R)$, chosen the sort k in `M` as our `State` subsort, and defined some state predicates Π along with their semantics in a suitable module, we implicitly have a Kripke structure $\mathcal{K}(\mathcal{R}, k)_\Pi$ on the set of atomic propositions AP_Π. Now, given an initial state $[t] \in T(\Sigma/E, k)$ and an LTL formula $\varphi \in \text{LTL}(AP_\Pi)$ we would like to have a procedure to decide the satisfaction relation:

$$\mathcal{K}(\mathcal{R}, k)_\Pi, [t] \;\models \varphi$$

Unfortunately, this relation is undecidable, unless these two conditions hold [CDE+16]:

1. the set of states in $T(\Sigma/E, k)$ that are *reachable* from $[t]$ by rewriting is *finite*

2. the rewrite theory $\mathcal{R} = (\Sigma, E, \phi, R)$ specified by `M` plus the equations D defining the predicates Π are such that:

   - both E and $E \cup D$ are (ground) Church-Rosser and terminating, perhaps modulo some axioms A, with $(\Sigma, E) \subseteq (\Sigma \cup \Pi, E \cup D)$ being a protecting extension.
   - R is (ground) coherent relative to E (again, perhaps modulo some axioms A).

Under the above assumptions, both the state predicates Π and the transition relation $\rightarrow^1_{\mathcal{R}}$ are *computable* and, given the infinite reachability assumption, we can settle the above satisfaction problem using a *model-checking procedure* [CDE+16].

For this, Maude uses an on-the-fly LTL model-checking procedure that works as follows. Each LTL formula φ has an associated Büchi automaton B_φ whose acceptance ω-language is exactly that of the behaviors satisfying φ. We can reduce the satisfaction problem to the *emptiness problem* of the language accepted by the *synchronous* product of $B_{\neg\varphi}$ and (the Büchi automaton associated with) $\mathcal{R} = (\Sigma, E, \phi, R)$. The formula φ is satisfied if and only if such language is empty. The model-checking procedure checks emptiness by looking for a counterexample, that is, an infinite computation belonging to the language recognized by the synchronous product [CDE+16].

CHAPTER 3
THE KEY2PHONE SYSTEM

In this chapter, we introduce the Key2phone system [CLB$^+$15]. This model was designed trying to meet all the security requirements pointed by the group of experts that were asked to analyze the system. In [CLB$^+$15], FSP (Finite State Process) and LTS(Labeled Transition Systems) are used to formalize the specification, to increase the formal guarantees of the system as well as their understanding and to emphasize the evolutionary nature of the system requirements. In the following section, we can summarize the list of threats and vulnerabilities discussed in [CLB$^+$15].

3.1 Security threats and vulnerabilities

In this section we recall the list of security threats and vulnerabilities of the Key2phone system. Every subsection explains each known vulnerability and the measures that must be taken in order to solve it.

3.1.1 Human physical safety

During an emergency situation, e.g., fire breaks into the building, is very inconvenient for a user to dial the identifier of the door to open. A safety measure for this is to use different sensors that trigger when a danger is detected and automatically open the doors. However, it is necessary to avoid false alarms.

Also, people can be trapped in-between or under the doors while closing them. To avoid this, it is advisable to install door entrapment protection mechanism.

3.1.2 Follow on attack or tailgating

An unauthorized person may enter the premises while the door is still closing. Using an automatic locking system can be a good counter measure.

3.1.3 Lost, theft, misplace of mobile phone

There is a high risk of misplacing the smartphone, which can expose the system to unauthorized access. As a counteract measure, an authentication mechanism should be used. Also, reset mechanisms can be implemented in case that the user forgets his/her credentials.

3.1.4 Password cracking

Even if the smartphone is secured with authentication mechanisms, the attackers can use a series of different techniques in order to crack the password. To avoid this, three actions can be applied; (i) increasing the entropy of the password by using uncommon and lengthy compositions; (ii) blocking any user after three failed attempts of authentication; and (iii) adding a delay of 5 seconds for every failed attempt.

3.1.5 Bluetooth hacked

When the Bluetooth functionality is activated, hackers can intercept the signal and gain full access to the smartphone. As a counteract measure, it is advisable that the discovery and connect mode are always turned off; also, the verifier should not accept unknown claims.

3.1.6 Spoofing

Attackers can employ *caller ID spoofing* to fake the authorized mobile number to open the door. Simply authenticating the caller as mentioned in Subsection 3.1.3 before opening the door can protect against *spoofing*.

3.1.7 Attacks during data transmission

In order to avoid attackers to steal data packets travelling over network, it is recommendable to use end-to-end protection and Cellular Message Encryption Algorithms (CMEA) [DW97].

3.1.8 Disruption of services to authorized users

Attackers can employ techniques to flood the system to prevent a legitimate user from accessing the service. A mechanism for integrity management, intrusion or anomaly detection systems, and timeliness detection of data can be implemented as counter measures.

3.1.9 Attacks by authorized users

Theres is a chance that a user can misuse any system or device, either intentionally or by accident, but designing suitable policies for the authorized user and ensuring that they all understand and adhere to the policies can help to prevent from accidental misuse.

3.1.10 Phising

Hackers can obtain the password by technical subterfuge, or by using social engineering to obtain it directly from the user. The solution for this is similar to the one exposed in 3.1.9.

3.1.11 Software/hardware vulnerabilities

Attackers may exploit vulnerabilities in both software and hardware. The only thing against this is using high quality hardware and software.

3.2 Key2phone system design

The formalization of the Key2phone system begins with the *authorization function*, which is related to vulnerability 3.1.9. This involves the creation of a log where every user's activity will be registered in order to guarantee *accountability*.

For the authorization, two kinds of identifiers are used: the *primary key* and the *non-primary key*. The primary key bearers are those who have administrator rights, while the non-primary key holders are general users. The Key2phone system is a composition of six different modes:

- *Management mode*
- *Move mode*
- *Remote mode*
- *Sleep mode*
- *Normal mode*
- *Emergency mode*

Non-primary key users only have access to the *Normal mode*, *Move mode*, and *Emergency mode*. On the other hand, primary key users have authorization to all modes. Figures 3.1 and 3.2 illustrate which modes can access the different users.

Management mode. This mode is only reachable by the primary key holders, i.e., administrators. In this mode, the user can (i) add any new non-primary users; (ii) update or delete the existing non-primary users; (iii) download the activity log; and (iv) configure security rules through the *configuration management system*. These security rules are related to the *Sleep mode* and the *Remote mode* and are used to specify, for example, at what time the *Sleep mode* should be enabled or disabled.

There is a demand to keep apart administrators from general users, but in order to correctly identify different users and to prevent any misuse related to threat of Section 3.1.3, user authentication is required. This can be achieved by using an username-password pair that is simple to implement and does not add extra costs. However, there exists the risk identified in Section 3.1.4 and a delay between failed attempts is also convenient. The idea is that, whenever somebody employs techniques like *dictionary attack*, *brute-force attack*, and/or *password-guess* for password cracking, the attacker will have to wait for the failed delay, thus, forcing them to spend more time for the task.

Move mode. This mode is activated when a user requires frequent door opening, allowing the user to open the door by just pressing a button. The door opens when a valid Bluetooth comes into the door proximity and open-button in the Key2phone Bluetooth application is pressed. This avoid users the tiresome task of entering the PIN code each time they open the door.

The *Move mode* can be accessed by every user after entering the correct PIN. In contrast to the user authentication mechanism, there is no increasing waiting time policy after a failed attempt. Instead, the number of attempts is limited to three. After that, the phone will be blocked from opening a door or changing the mode. This is not a definitive measure because an administrator can unlock the number via the *web-based configuration management system*. There is an implicit danger in using this measure, as it can be abused to block someones phone intentionally. However, in this case the attacker needs to authenticate as the user he or she wants to block, so it's a rare case that can hardly happen. This kind of mechanisms will also prevent *caller ID spoofing*, i.e., vulnerability in Section 3.1.6, since fake caller will fail to open the door without a valid PIN code. Also, the objective of any Denial of service (DoS) or Distributed Denial of Service (DDoS) attack is to make unavailable any service by flooding the system with superfluous request, but this kind of attack will be shutdown due to the blocking of any number that fails to give a correct PIN in three attempts. This can be seen as a mitigation for threat in Section 3.1.8.

Once the PIN is correct and the user enters the *Move mode*, the mobile phone's Bluetooth is also set discoverable and connectable. To counteract vulnerability in Section 3.1.7, a time has to be set to for which the *Move mode* has to be enabled. This way the phone will be protected against Bluetooth hacking and the user will be relieved from having to separately press a button for enabling the Bluetooth.

Sleep mode. This mode defines the operational behavior of the door-lock system during the night time. Once the *Sleep mode* is activated, the rules determined by the primary key holder get activated; for example, the time after which the door has to be locked with no more operations.

Remote mode. In this mode, the administrator has the ability to control the door from remote locations. When this mode is accessed, the rules defined by the administrator are loaded and taken into use. For the functioning of the *Remote mode*, it is necessary to detect the administrator's location. Also, the administrator can activate the geo-location service, which will continuously record the current location and will enable remote mode when the administrator is out of the system premises.

Normal mode. This mode is the default mode for opening the doors. The access to this mode is also via PIN code, and the three attempts policy is applied. In this mode an identified user can dial the number of the door or reach the door proximity with Bluetooth enabled to open the door.

Emergency mode. This mode is thought for the case when there is some kind of natural disaster or danger within the building. When this mode is activated, the system notifies people inside the premise and opens the door automatically to facilitate an escape route in a short time. This way, it is solved the problem identified in vulnerability in Section 3.1.1. To correctly operate the *Emergency mode*, it requires additional sensors, e.g., smoke sensors, to be deployed and connected with the Key2phone system. However, the sensor-system has to protect from nuisance or false alarms and, at the same time, it must trigger when there is any true cause. A sensor that automatically adjusts the sensitivity without affecting its performance during no-alarm situations can prove to be extremely helpful.

In order to ensure people's safety using this system, and to avoid vulnerability in Section 3.1.1, it is necessary to protect any entrapment in between the doors. Likewise, for every open-door operation a close-door operation is followed that can be helpful to prevent tailgating, which is a solution for the vulnerability of Section 3.1.2.

Similarly to [CLB$^+$15], no measure is included against vulnerabilities of Sections 3.1.7, 3.1.10, and 3.1.11, because they are related to the data transmission, people actions, and employed hardware/software, which are either out of scope or need a deeper study.

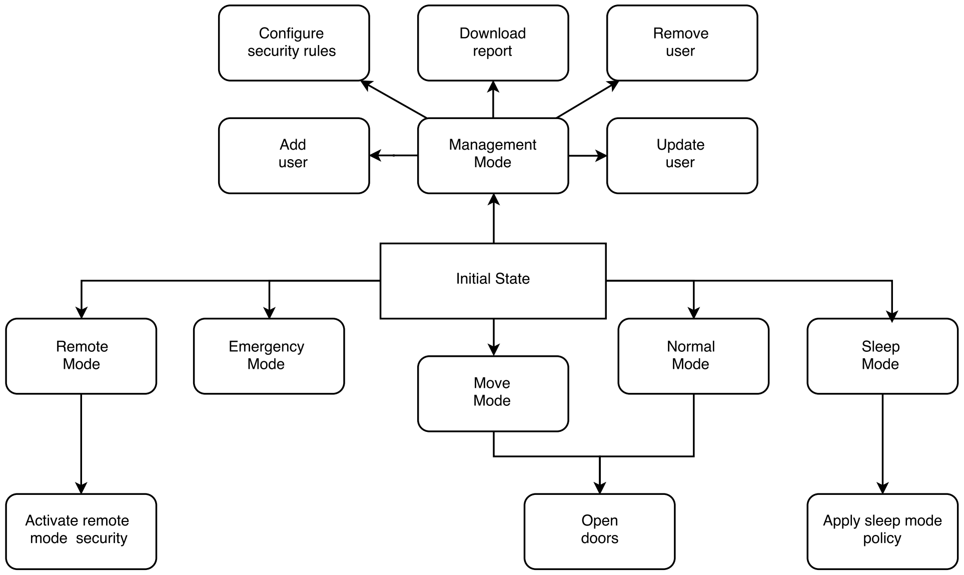


Figure 3.1: Admin action flow

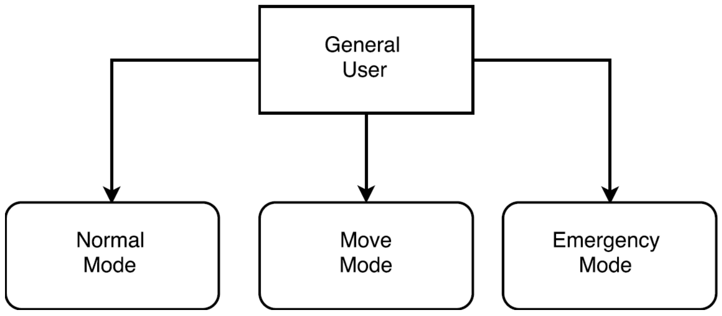


Figure 3.2: General user action flow

CHAPTER 4

FORMAL SPECIFICATION OF THE KEY2PHONE SYSTEM USING MAUDE

As already mentioned, the goal of this thesis is to verify safety and liveness properties of the Key2phone system by means of a model formalization and analysis using *Maude*, based on the guidelines in [CLB$^+$15]. In order to become familiar with and gain some mastery of the system, we first use the online stepper and analysis tool Anima [ABFS15], which effectively supports reachability analysis in a visual manner.

The Anima tool offers a rich and highly dynamic, parameterized technique for trace of inspection in rewriting logic theories that allows the non-deterministic execution of a given conditional rewrite theory to be followed up in different ways [ABFS15]. This tool allows us to inspect the computation tree for a given input expression so that any state that is not supposed to be there can be easily identified. Wherever an anomalous system configuration tree is detected, we can inspect, browse, query, search, and/or slice the computation to locate coding mistakes.

We needed to represent the possible changes that the system will experiment during an execution. For this, we defined that the parameters that we wanted to monitor were:

- The identity of the user.
- The current mode of the system.

- A counter to check the login attempts.
- The door that is being used at each moment.
- The action the user is doing.
- The information that will be uploaded to the log.

With this in mind, we created a constructor that encapsulates all this information, which is defined as follows:

```
op <_|_|_|_|_|_|_> :  Key Mode Int Pin Door Action Log -> Status .
```

The above operator has seven arguments, which are explained as follows:

- The first parameter admits terms of sort `Key` and represents the category of the user: primary key holders are the ones with administrator privileges, and to represent them is used the value `primary`; non-primary key holders, or general users, are those who are not administrators and the value `nonprimary` is used to represent them; finally the constant `nouser`, indicates that the user has not authenticated himself/herself yet.
- The second parameter accepts terms of sort `Mode` and stores the actual mode in which the user is. This parameter has nine different values, six of them are the modes mentioned in Chapter 3, while the three remaining ones are used to represent the user's state. These states are: `unlogged`, `logged` and `blocked`. The two first are transition modes, in which the user has not authenticated himself, or the user has identified himself but hasn't chosen any mode, respectively. The `blocked` constant is used to represent the case when the user could not authenticate himself in three attempts, and an administrator is required to unlock the account.
- The third parameter's only purpose is to count the authentication attempts a user has made and change the mode to `blocked` after too many attempts have happened. It accepts terms of sort `Int`.
- The fourth parameter accepts terms of sort `Pin`. Here we can see as list of the different terms that are seen in this parameter:

- *nopin*: This value means that there is no need to authenticate the user.
- *trypin*: This value indicates that the user is attempting to authenticate himself/herself.
- *correctpin*: The PIN is correct and access to the desired mode is granted.
- *incorrectpin*: The PIN is incorrect and the user needs to try a new attempt, unless there are no more attempts left and the phone number is blocked.

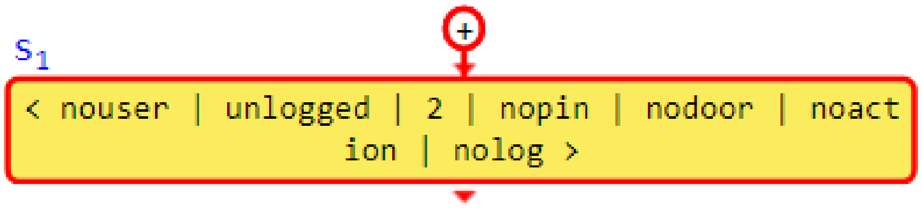
- For the fifth parameter, terms of sort `Door` are accepted and is used to represent the sensor in the entrapment of the door. This parameter indicates if the door is opened, closed or if there's an object thats does not let the door to be closed.

- The sixth parameter accepts terms of sort `Action` and records if there is an action running and which action is.

- The seventh parameter admits terms of sort `Log`, which store the information of the system's log. Usually, it is a pair that is composed of a key and an action/door opened. In this work we do not consider the way in which the log is updated since it is irrelevant for our goal.

Figure 4.1 shows the predefined configuration (of sort `Status`) that represents the initial system state, i.e., when the user just started the application and has not done anything else.

Figure 4.1: Initial state of the system



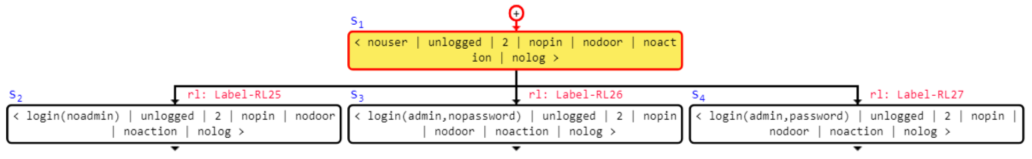
```
< nouser | unlogged | 2 | nopin | nodoor | noact
ion | nolog >
```

The next state stemming from this state is the authentication of the user. In order to represent this, the following transition rules have been specified:

```
rl < nouser | unlogged | N | nopin | nodoor | noaction | nolog > =>
< login(admin,password ) | unlogged | N | nopin | nodoor | noaction
| nolog > .

rl < nouser | unlogged | N | nopin | nodoor | noaction | nolog > =>
< login(admin,nopassword) | unlogged | N | nopin | nodoor | noaction
| nolog > .

rl < nouser | unlogged | N | nopin | nodoor | noaction | nolog > =>
< login(noadmin) | unlogged | N | nopin | nodoor | noaction | nolog >.
```

This set of rules specifies the change in the first parameter. Here we represent all the three possible states to which the system can evolve. The first one represents a primary key bearer who identifies himself/herself by using a correct pair username/password; the second one is the same scenario, but the user types an incorrect password; the last state corresponds to a general user who authenticates itself. A graphical representation of all the possible one-step transitions from the initial state can be seen in Figure 4.2.

Figure 4.2: Login transitions


S1
< nouser | unlogged | 2 | nopin | nodoor | noaction | nolog >
rl: Label-RL25
rl: Label-RL26
rl: Label-RL27
S2
< login(noadmin) | unlogged | 2 | nopin | nodoor | noaction | nolog >
S3
< login(admin,nopassword) | unlogged | 2 | nopin | nodoor | noaction | nolog >
S4
< login(admin,password) | unlogged | 2 | nopin | nodoor | noaction | nolog >

Note that the non-primary key bearers do not need a password because the administrators keep a list of general users that can regain access to the system. This section is made to entirely match the initial state instead of only matching the `nouser` constant, in order to avoid that any blocked user attempts to re-login himself/herself.

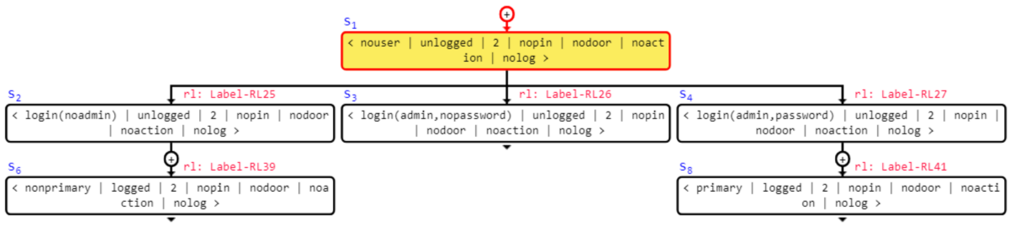
For this system specification, the three-attempt authentication mechanism is implemented. The following rules define its behavior:

```
rl < login(admin,password) | unlogged | N | nopin | nodoor | noaction
```

```
| nolog > =>
< primary | logged | counter | nopin | nodoor | noaction | nolog >.

rl < login ( noadmin ) | unlogged | N | nopin | nodoor | noaction
| nolog > =>
< nonprimary | logged | counter | nopin | nodoor | noaction | nolog >.
```

The above set of rules establishes how, whenever the username/password pair is correct, the *Key* parameter of `Status` is changed to `primary` or to `non-primary`, depending on the user category. An example of a correct login is illustrated in Figure 4.3

Figure 4.3: Correct login transition


S1
< nouser | unlogged | 2 | nopin | nodoor | noaction | nolog >
rl: Label-RL25
rl: Label-RL26
rl: Label-RL27
S2
< login(noadmin) | unlogged | 2 | nopin | nodoor | noaction | nolog >
S3
< login(admin,nopassword) | unlogged | 2 | nopin | nodoor | noaction | nolog >
S4
< login(admin,password) | unlogged | 2 | nopin | nodoor | noaction | nolog >
rl: Label-RL39
rl: Label-RL41
S6
< nonprimary | logged | 2 | nopin | nodoor | noaction | nolog >
S8
< primary | logged | 2 | nopin | nodoor | noaction | nolog >

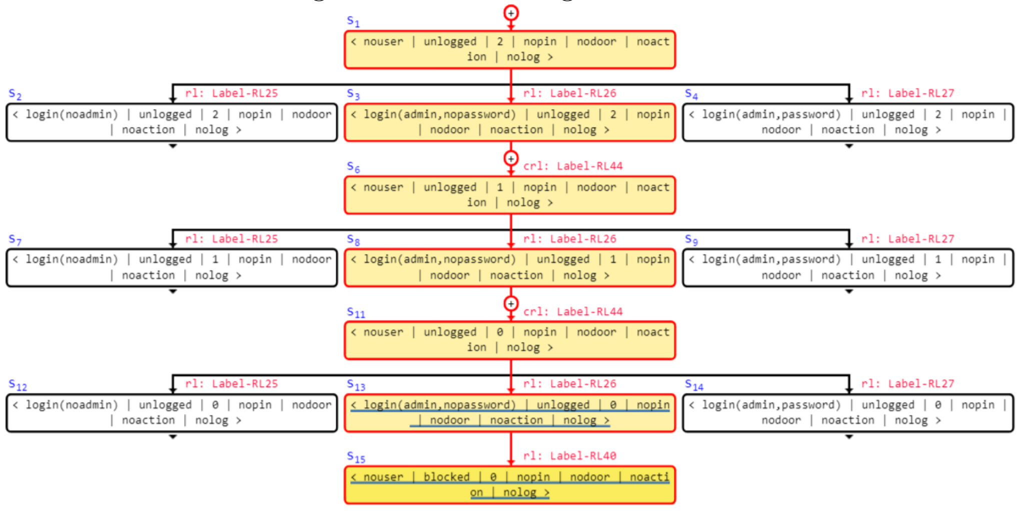
In the case when the combination username/password is incorrect, the following rules apply:

```
crl < login ( admin , nopassword ) | unlogged | N | nopin | nodoor | noaction
| nolog > =>
< nouser | unlogged | N - 1 | nopin | nodoor | noaction | nolog >
if ( N > 0 ) .

rl < nouser | unlogged | 0 | nopin | nodoor | noaction | nolog > =>
< nouser | blocked | 0 | nopin | nodoor | noaction | nolog > .
```

This rules are designed in a way that the user is taken back to the `nouser` state so he/she can try to authenticate again, provided there are remaining attempts left. Figure 4.4 illustrate the system transitions when a user fails to authenticate himself/herself in three attempts.

Due to the limitations for general users in terms of accessing the different modes, without loss of generality, in this work only the branch of the

Figure 4.4: Failed login transition

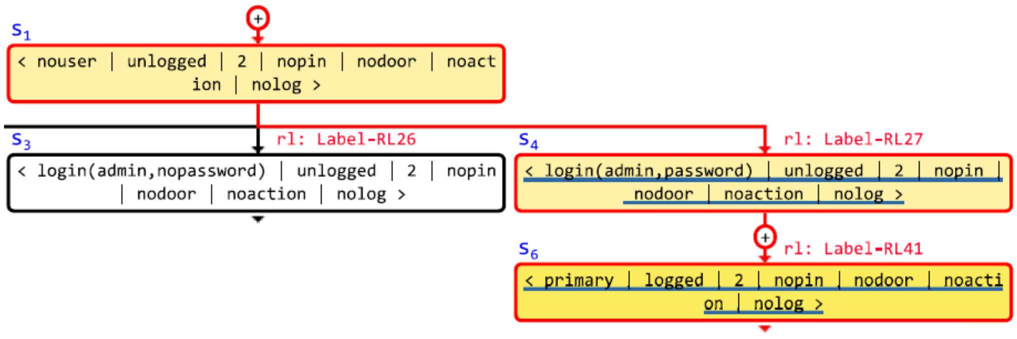


S1
< nouser | unlogged | 2 | nopin | nodoor | noaction | nolog >
rl: Label-RL25
rl: Label-RL26
rl: Label-RL27
S2
< login(noadmin) | unlogged | 2 | nopin | nodoor | noaction | nolog >
S3
< login(admin,nopassword) | unlogged | 2 | nopin | nodoor | noaction | nolog >
S4
< login(admin,password) | unlogged | 2 | nopin | nodoor | noaction | nolog >
crl: Label-RL44
S6
< nouser | unlogged | 1 | nopin | nodoor | noaction | nolog >
S7
< login(noadmin) | unlogged | 1 | nopin | nodoor | noaction | nolog >
S8
< login(admin,nopassword) | unlogged | 1 | nopin | nodoor | noaction | nolog >
S9
< login(admin,password) | unlogged | 1 | nopin | nodoor | noaction | nolog >
crl: Label-RL44
S11
< nouser | unlogged | 0 | nopin | nodoor | noaction | nolog >
S12
< login(noadmin) | unlogged | 0 | nopin | nodoor | noaction | nolog >
S13
< login(admin,nopassword) | unlogged | 0 | nopin | nodoor | noaction | nolog >
S14
< login(admin,password) | unlogged | 0 | nopin | nodoor | noaction | nolog >
rl: Label-RL40
S15
< nouser | blocked | 0 | nopin | nodoor | noaction | nolog >

search related to the primary key holders is discussed since the other ones are perfectly similar.

Once the user succeeds at the authentication phase, the system proceeds to the status shown in Figure 4.5, where the *key* is `primary` and the *mode* enters the transition value `logged`.

Figure 4.5: Primary key authentication



S1
< nouser | unlogged | 2 | nopin | nodoor | noaction | nolog >
S3
rl: Label-RL26
< login(admin,nopassword) | unlogged | 2 | nopin | nodoor | noaction | nolog >
S4
rl: Label-RL27
< login(admin,password) | unlogged | 2 | nopin | nodoor | noaction | nolog >
S6
rl: Label-RL41
< primary | logged | 2 | nopin | nodoor | noaction | nolog >

At this point, the next thing a user needs to do is to select a mode. Since we are representing the case of an administrator, he/she has full access to all six available modes, as we can see in Figure 4.6. In order to represent this in our model, the following transition rules are provided:

```
< primary | logged | N | nopin | nodoor | noaction | nolog > =>
    < primary | management | N | nopin | nodoor | noaction | nolog > .

rl < primary | logged | N | nopin | nodoor | noaction | nolog > =>
    < primary | emergency | N | nopin | nodoor | noaction | nolog > .

rl < primary | logged | N | nopin | nodoor | noaction | nolog > =>
    < primary | move | N | nopin | nodoor | noaction | nolog > .

rl < primary | logged | N | nopin | nodoor | noaction | nolog > =>
    < primary | remote | N | nopin | nodoor | noaction | nolog > .

rl < primary | logged | N | nopin | nodoor | noaction | nolog > =>
    < primary | sleep | N | nopin | nodoor | noaction | nolog > .

rl < primary | logged | N | nopin | nodoor | noaction | nolog > =>
    < primary | normal | N | nopin | nodoor | noaction | nolog > .
```

Non-primary user's code is exactly the same, changing the key and restricting the mode to the ones he/she can enter:

```
rl < nonprimary | logged | N | nopin | nodoor | noaction | nolog > =>
    < nonprimary | emergency | N | nopin | nodoor | noaction | nolog > .

rl < nonprimary | logged | N | nopin | nodoor | noaction | nolog > =>
    < nonprimary | move | N | nopin | nodoor | noaction | nolog > .

rl < nonprimary | logged | N | nopin | nodoor | noaction | nolog > =>
    < nonprimary | normal | N | nopin | nodoor | noaction | nolog > .
```

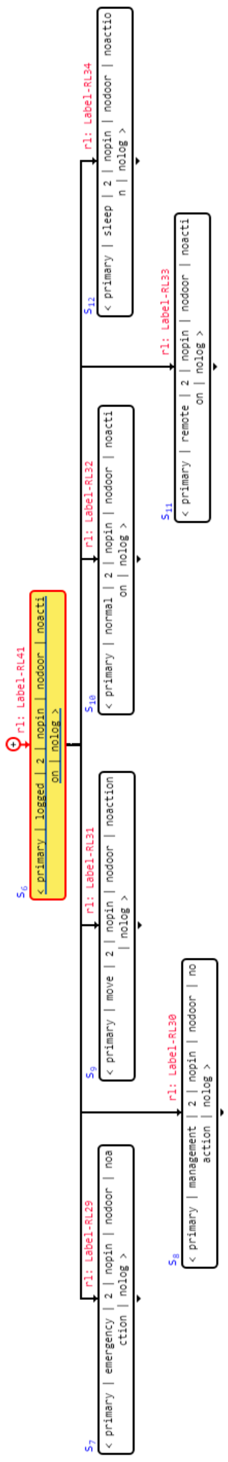

s6
r1: Label-RL41
< primary | logged | 2 | nopin | nodoor | noaction | nolog >
s7
r1: Label-RL29
< primary | emergency | 2 | nopin | nodoor | noaction | nolog >
s8
r1: Label-RL30
< primary | management | 2 | nopin | nodoor | noaction | nolog >
s9
r1: Label-RL31
< primary | move | 2 | nopin | nodoor | noaction | nolog >
s10
r1: Label-RL32
< primary | normal | 2 | nopin | nodoor | noaction | nolog >
s11
r1: Label-RL33
< primary | remote | 2 | nopin | nodoor | noaction | nolog >
s12
r1: Label-RL34
< primary | sleep | 2 | nopin | nodoor | noaction | nolog >

Figure 4.6: Mode entering

Similarly to the login phase, we need to make a full matching of the state to avoid general users entering modes that only apply for administrators.

The first mode to inspect is the *Management mode*. The system first changes the term `noaction` to `actionexpected`. This is the way to represent the state where the user has not chosen any action. When the user finally executes an action, it is recorded in the sixth parameter and the log is updated. The following rules are used to implement this behavior:

```
rl < primary | management | N | nopin | nodoor | noaction | nolog > =>
< primary | management | N | nopin | nodoor | actionexpected | nolog > .

rl < primary | management | N | nopin | nodoor | MACTION | nolog > =>
< primary | management | N | nopin | nodoor | MACTION
| updatelog ( primary , MACTION ) >.
```

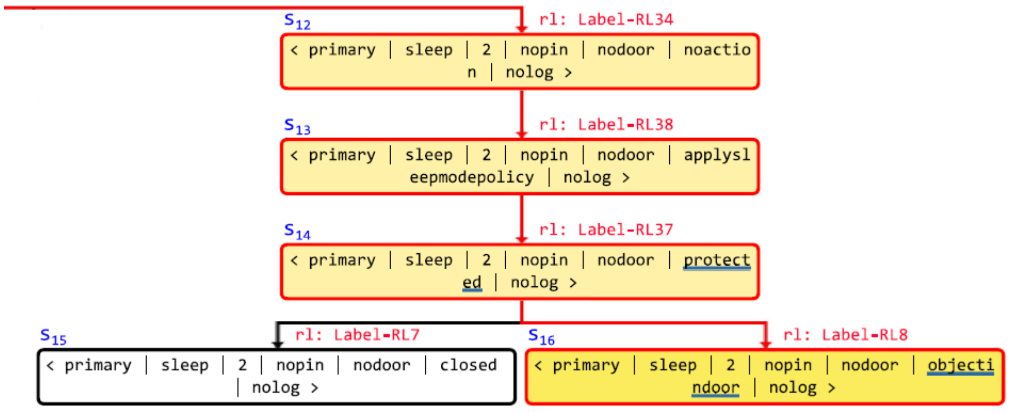
Where `MACTION` is a variable of sort `Maction` that has been previously defined in the program and is used to cover all the different actions that can be done in the management mode. The different actions are represented by using these transition rules:

```
rl actionexpected => adduser .
rl actionexpected => removeuser .
rl actionexpected => updateuser .
rl actionexpected => downloadreport .
rl actionexpected => configuresecurityrules .
```

The next mode to design and inspect is *Sleep mode*. When this mode is entered, all the security rules, previously defined by the administrator, are applied and the specified doors are closed. In Figure 4.7, the branching of the sleep mode section is given. There is a bifurcation at the end of the branch that represents when the sensors detect an object in the door threshold. This functionality is modeled by means of the following rules:

```
rl < primary | sleep | N | nopin | nodoor | noaction | nolog > =>
   < primary | sleep | N | nopin | nodoor | applysleepmodepolicy | nolog>.

rl < primary | sleep | N | nopin | nodoor | applysleepmodepolicy
| nolog> =>
   < primary | sleep | N | nopin | nodoor | protected | nolog > .
```

Figure 4.7: Sleep mode


s_{12}
rl: Label-RL34
< primary | sleep | 2 | nopin | nodoor | noaction | nolog >
s_{13}
rl: Label-RL38
< primary | sleep | 2 | nopin | nodoor | applysleepmodepolicy | nolog >
s_{14}
rl: Label-RL37
< primary | sleep | 2 | nopin | nodoor | protected | nolog >
s_{15}
rl: Label-RL7
< primary | sleep | 2 | nopin | nodoor | closed | nolog >
s_{16}
rl: Label-RL8
< primary | sleep | 2 | nopin | nodoor | objectindoor | nolog >

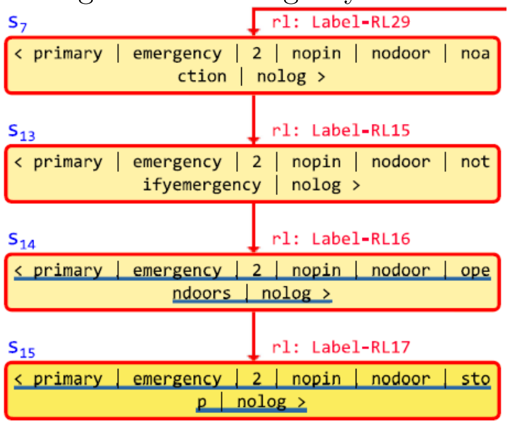
The next mode is the *Emergency mode*. This mode is designed to first notify all the users of the given danger and then open all the doors. Also, it enters in stop mode to avoid trapping someone. Here are the rules used that model this behavior:

```
rl < KEY | emergency | N | nopin | nodoor | noaction | nolog > =>
    < KEY | emergency | N | nopin | nodoor | notifyemergency | nolog > .

rl < KEY | emergency | N | nopin | nodoor | notifyemergency | nolog > =>
    < KEY | emergency | N | nopin | nodoor | opendoors | nolog > .

rl < KEY | emergency | N | nopin | nodoor | opendoors | nolog > =>
    < KEY | emergency | N | nopin | nodoor | stop | nolog > .
```

Note that, since there is a possibility of mortal danger, a variable `KEY` of sort `Key` is used. This way, the notification will be sent to everybody, disregarding if it is and administrator, a general user, or someone who has not authenticated himself/herself. A slice of the computational tree related to this mode can be seen in Figure 4.8.

The *Remote mode* is supposed to work when the location of the administrator indicates that he or she is no longer in the building. Since this is

Figure 4.8: Emergency mode


S7
rl: Label-RL29
< primary | emergency | 2 | nopin | nodoor | noaction | nolog >
S13
rl: Label-RL15
< primary | emergency | 2 | nopin | nodoor | notifyemergency | nolog >
S14
rl: Label-RL16
< primary | emergency | 2 | nopin | nodoor | opendoors | nolog >
S15
rl: Label-RL17
< primary | emergency | 2 | nopin | nodoor | stop | nolog >

irrelevant for this work, the entrance to this mode is the same as in the other modes:

```
rl < KEY | remote | N | nopin | nodoor | noaction | nolog > =>
< KEY | remote | N | nopin | nodoor | activateremotemodesecurity | nolog > .

rl < KEY | remote | N | nopin | nodoor | activateremotemodesecurity
| nolog > => < KEY | remote | N | nopin | nodoor | protected | nolog > .
```

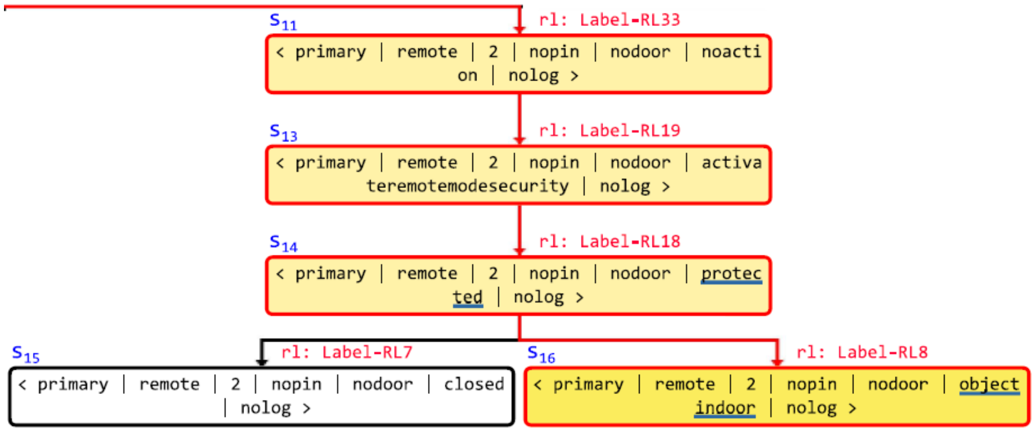
In this mode we also have the bifurcation explained above about the threshold sensor, as illustrated in Figure 4.9.

Now, only remains to explain *Normal mode* and *Move mode*. Both behave in a very similar way, the only difference being that in *Move mode* the user has to specify how long he is going to stay in that mode. Both modes require a PIN in order to regain access to them. The implementation of this mechanism is very similar to the login mechanism. First we apply the following transition rules to represent both a correct PIN and an incorrect one:

```
crl < KEY | PINM | N | trypin | nodoor | noaction | nolog > =>
< KEY | PINM | N | correctpin | nodoor | noaction | nolog > if ( N > 0 ) .

crl < KEY | PINM | N | trypin | nodoor | noaction | nolog > =>
< KEY | PINM | N | incorrectpin | nodoor | noaction | nolog > if ( N > 0) .
```

Then, the resulting states matches the left-hand side of the following con-

Figure 4.9: Remote mode


S11
rl: Label-RL33
< primary | remote | 2 | nopin | nodoor | noaction | nolog >
S13
rl: Label-RL19
< primary | remote | 2 | nopin | nodoor | activateremotemodesecurity | nolog >
S14
rl: Label-RL18
< primary | remote | 2 | nopin | nodoor | protected | nolog >
S15
rl: Label-RL7
< primary | remote | 2 | nopin | nodoor | closed | nolog >
S16
rl: Label-RL8
< primary | remote | 2 | nopin | nodoor | objectindoor | nolog >

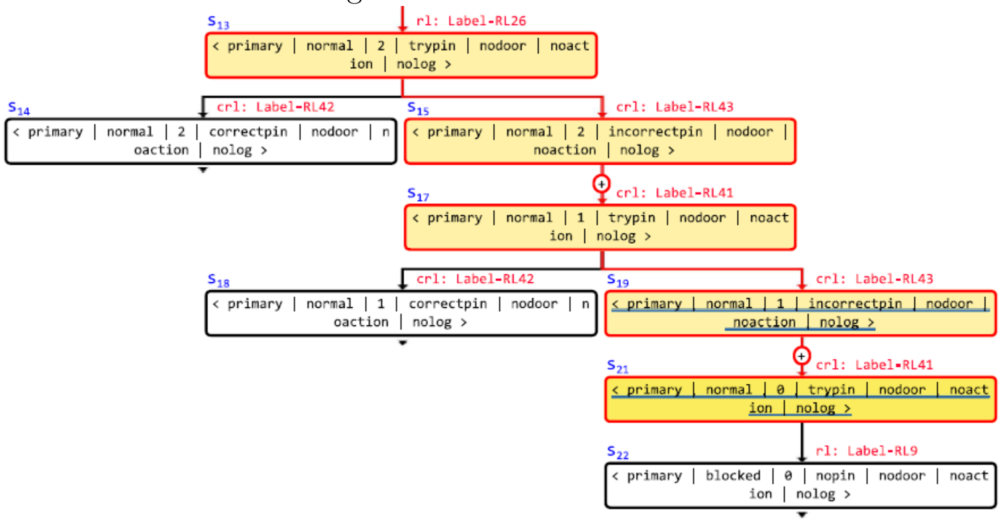
ditional rules:

```
crl < KEY | PINM | N | correctpin | nodoor | noaction | nolog > =>
< KEY | PINM | counter | correctpin | doorselection | noaction | nolog >
if ( N > 0 ) .

crl < KEY | PINM | N | incorrectpin | nodoor | noaction | nolog > =>
< KEY | PINM | N - 1 | trypin | nodoor | noaction | nolog >
if ( N > 0 ) .

rl < KEY | PINM | 0 | trypin | nodoor | noaction | nolog > =>
< KEY | blocked | 0 | nopin | nodoor | noaction | nolog > .
```

The variable `KEY` is similar as the one referred to in *Emergency mode* due to the fact that both key holders can have access to both modes. The variable `PINM` is used to refer to both modes in one sentence, yielding to a more compact and optimized code. Figure 4.10 illustrates the scenario where all three attempts fail.

When the PIN is correct, it grants the user access and the parameter `Door` changes its initial value `nodoor` to `doorselection`, which indicates the state where the system is waiting for the user to choose a door via dialing its ID number or get into the range of the Bluetooth. To represent the two actions, the following transition rules are given:

Figure 4.10: Incorrect PIN


S13
rl: Label-RL26
< primary | normal | 2 | trypin | nodoor | noaction | nolog >
S14
crl: Label-RL42
< primary | normal | 2 | correctpin | nodoor | noaction | nolog >
S15
crl: Label-RL43
< primary | normal | 2 | incorrectpin | nodoor | noaction | nolog >
S17
crl: Label-RL41
< primary | normal | 1 | trypin | nodoor | noaction | nolog >
S18
crl: Label-RL42
< primary | normal | 1 | correctpin | nodoor | noaction | nolog >
S19
crl: Label-RL43
< primary | normal | 1 | incorrectpin | nodoor | noaction | nolog >
S21
crl: Label-RL41
< primary | normal | 0 | trypin | nodoor | noaction | nolog >
S22
rl: Label-RL9
< primary | blocked | 0 | nopin | nodoor | noaction | nolog >

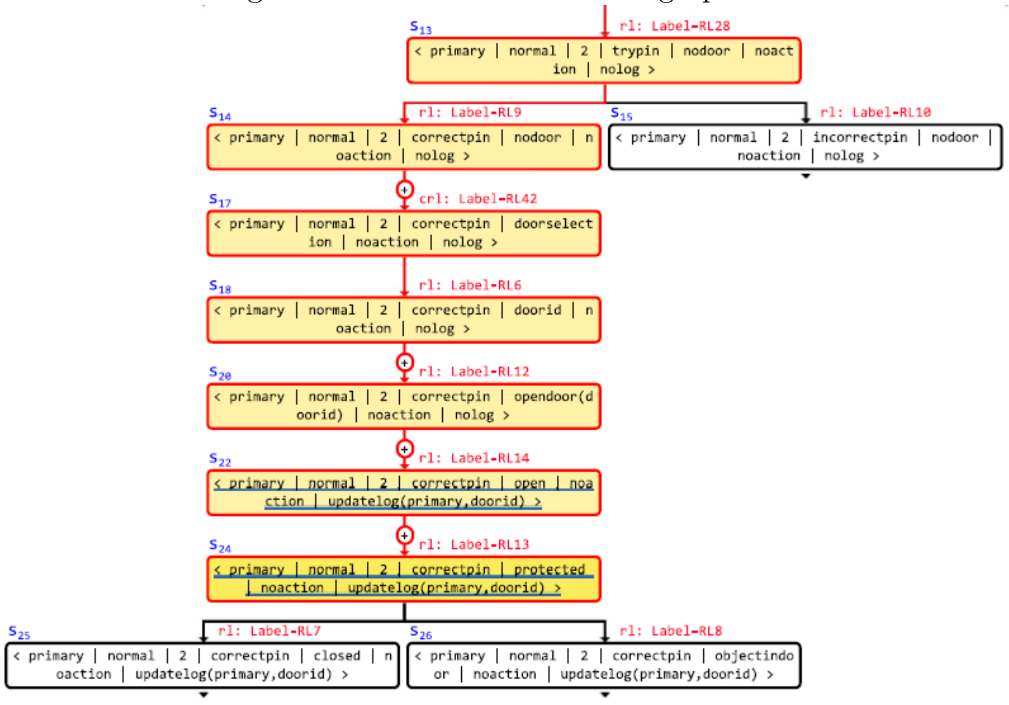
```
rl doorselection => doorid .
```

This changes the state so that it can match the following rules:

```
rl < KEY | PINM | N | correctpin | doorid | noaction | nolog > =>
< KEY | PINM | counter | correctpin | opendoor ( doorid ) | noaction
| nolog > .

rl < KEY | PINM | N | correctpin | opendoor ( doorid ) | noaction | nolog >
=>
< KEY | PINM | counter | correctpin | (doorid , open ) | noaction
| updatelog ( KEY , doorid ) > .

rl < KEY | PINM | counter | correctpin | (doorid , open ) | noaction
| updatelog ( KEY , doorid ) > =>
< KEY | PINM | counter | correctpin | (doorid , protected ) | noaction
| updatelog ( KEY , doorid ) > .
```

Once the door is opened, the fifth parameter stores the number of the door and the actual status. Also, the information is stored in the system's log by using the term `updatelog(KEY,doorid)`. A graphical representation of the whole transition can be seen in Figure 4.11.

Figure 4.11: Correct PIN and log update


S13
rl: Label-RL28
< primary | normal | 2 | trypin | nodoor | noaction | nolog >
S14
rl: Label-RL9
< primary | normal | 2 | correctpin | nodoor | noaction | nolog >
S15
rl: Label-RL10
< primary | normal | 2 | incorrectpin | nodoor | noaction | nolog >
S17
crl: Label-RL42
< primary | normal | 2 | correctpin | doorselection | noaction | nolog >
S18
rl: Label-RL6
< primary | normal | 2 | correctpin | doorid | noaction | nolog >
S20
rl: Label-RL12
< primary | normal | 2 | correctpin | opendoor(doorid) | noaction | nolog >
S22
rl: Label-RL14
< primary | normal | 2 | correctpin | open | noaction | updatelog(primary,doorid) >
S24
rl: Label-RL13
< primary | normal | 2 | correctpin | protected | noaction | updatelog(primary,doorid) >
S25
rl: Label-RL7
< primary | normal | 2 | correctpin | closed | noaction | updatelog(primary,doorid) >
S26
rl: Label-RL8
< primary | normal | 2 | correctpin | objectindoor | noaction | updatelog(primary,doorid) >

CHAPTER 5 KEY2PHONE ANALYSIS USING MAUDE

Every property of a system is an intersection of a safety property and a liveness property. Properties, which have long been used for reasoning about systems, are *sets of traces* (i.e., system executions). Hyperproperties can express security policies, such as confidentiality, that properties cannot. However, hyperproperties cannot be verified by *model-checking* as they depend on an infinite set of infinite sets of system executions [CS10]. This is why in this master's thesis we focus on the classical *system properties* of safety and liveness which any system property can be reduced.

We have developed a *system module* that defines a rewrite theory that specifies the behavior of the Key2phone system. In Section 5.1 we provide some examples of reachability property analysis and discuss the results of the reachability tests. Then, in Section 5.2 we explain the properties checked for the system, as well as the results obtained of each one.

5.1 Reachability analysis

The aim of the reachability analysis is to ensure that all the intended states are reached by the system and, also, that no harmful state is ever reached. Thanks to the Anima tool [ABFS15], we can visually inspect that the intended states are met, but for a further verification, we used the Maude's `search` command.

The first search we made is related to the keys. We wanted to be sure that no general user is able to ever change its own key, unless he or she re-logins. We can see below the command used and the result of it.

```
 search in KEY2PHONE-CHECK :
< nonprimary | logged | counter | nopin | nodoor | noaction | nolog >
 =>*
< primary | A:Mode | B:Int | C:Pin | D:Door | E:Actions | F:Log > .

No solution.
states: 40  rewrites: 73 in 0ms real (~ rewrites/second)
```

To check this, we made a search with every other parameter as a variable, because we are only interested in checking that the first parameter never changes to primary. As we can see, the `search` command returns that there is no solution for this search, so it is verified that a user cannot change its own key without a re-login (i.e., re-executing the program).

For the next analysis, we want to check that no general user is able to access modes reserved only to the administrators. Below we can see the associated `search` execution and its result.

```
search in KEY2PHONE-CHECK :
< nonprimary | logged | counter | nopin | nodoor | noaction | nolog >
 =>*
 < A:Key | X:Adminmode | B:Int | C:Pin | D:Door | E:Actions | F:Log > .

No solution.
states: 40  rewrites: 73 in 1ms real (~ rewrites/second)
```

For this command, we used `X:Adminmode` as a variable that refers to all the reserved modes. As we can see, there is no solution and so, it is verified that none of the reserved modes can be accessed by a general user.

Now that we know that the general users can not change its own key, nor access the reserved modes for the administrators, we wanted to check that the general users can access the modes that it were designed for. Below we can see the commands used and the results of them.

```
 search in KEY2PHONE-CHECK :
< nonprimary | logged | counter | nopin | nodoor | noaction | nolog >
=>*
< A:Key | X:Mode | B:Int | C:Pin | D:Door | E:Actions |
F:Log > .

Solution 1 (state 0)
states: 1  rewrites: 1 in 2ms real (~ rewrites/second)
A:Key --> nonprimary
X:Mode --> logged
B:Int --> 2
C:Pin --> nopin
D:Door --> nodoor
E:Actions --> noaction
F:Log --> nolog

Solution 2 (state 1)
states: 2  rewrites: 2 in cpu 25ms real (~ rewrites/second)
A:Key --> nonprimary
X:Mode --> emergency
B:Int --> 2
C:Pin --> nopin
D:Door --> nodoor
E:Actions --> noaction
F:Log --> nolog

Solution 3 (state 2)
states: 3  rewrites: 3 in cpu 41ms real (~ rewrites/second)
A:Key --> nonprimary
X:Mode --> move
B:Int --> 2
C:Pin --> nopin
D:Door --> nodoor
E:Actions --> noaction
F:Log --> nolog

Solution 4 (state 3)
states: 4  rewrites: 4 in 54ms real (~ rewrites/second)
A:Key --> nonprimary
X:Mode --> normal
B:Int --> 2
C:Pin --> nopin
D:Door --> nodoor
```

```
E:Actions --> noaction
F:Log --> nolog

...

Solution 40 (state 39)
states: 40  rewrites: 73 in cpu 493ms real (~ rewrites/second)
A:Key --> nonprimary
X:Mode --> normal
B:Int --> 2
C:Pin --> correctpin
D:Door --> objectindoor
E:Actions --> noaction
F:Log --> updatelog(nonprimary,doorid)

No more solutions.
states: 40  rewrites: 73 in 504ms cpu (504ms real) (~ rewrites/second)
```

For this search, we omitted the solutions 5 trough 39 due to the duplication of the modes. As in the last search, we used a variable, the `X:Mode`, but this time to refer to all the modes that can be reached from the initial state. Disregarding the first result because is the starting point, it is verified that the system can reach all the intended states.

5.2 Model cheking

In the above section, we verified the reachability of the system with a few examples, but for a further study of the system's properties we need to make use of another tool. In order to properly achieve this aim and provide an accurate study of the desired properties, we make use of the LTL model checker of `Maude`.

As explained in Section 2.3.2, the module `KEY2PHONE-PREDS` contains the protected `system module` specification and includes the module `SATISFACTION` previously loaded into the `Maude` environment.

```
mod KEY2PHONE-PREDS is
protecting KEY2PHONE .
including SATISFACTION .
```

The next step is to choose a sort in the system specification and declare it as a subsort of the sort `State`. As we defined in the `KEY2PHONE` module, the intended sort for this is the `Status` sort because it contains all the information about the state of the system and is where all the changes will be made. With this decision, the next line is added to the file:

```
 subsort Status < State .
```

This module is used to monitor all the changes in `Status` so we defined the following variables:

```
var K : Key .
var M : Mode .
var I : Int .
var P : Pin .
var D : Door .
var AC : Actions .
var L : Log .
var S : Status .
```

that are used later to create *state predicates* that tell us when a desired parameters takes a value that we want to check, while disregarding the others.

Another module has been created, the `KEY2PHONE-CHECK` module, which is simply used to specify the initial system state where properties are to be checked.

```
mod KEY2PHONE-CHECK is
 protecting KEY2PHONE-PREDS .
 including MODEL-CHECKER .
 including LTL-SIMPLIFIER .

 ops initial1 : -> Status .

 eq initial1 = < nouser | unlogged | counter | nopin | nodoor
| noaction | nolog > .
```

The initial state defined in `KEY2PHONE-CHECK` is the same initial state in the system module because we want to make sure that the properties are true for every possible execution, and also, by declaring the initial state here, we do not haver to type the whole status every time we want to check a property.

From this point on, the next step is to define the properties we are interested to check. As stated in [CS10], all properties can be constructed as the intersection of a safety property and a liveness property, where:

- a *safety property* is a property that proscribes "bad things" and can be proved using an invariance argument, and
- a *liveness property* is a trace property that prescribes "good things" and can be proved using a well-foundedness argument.

As already mentioned, this system is not complex and a reachability test may suffice to check many interesting properties, but the problem rises when we want to make a conditional or hybrid check involving liveness properties of any kind as well.

5.2.1 Verified properties

Safety property ensure that nothing bad will happen, and the first property we want to check is related to this. According to the system's design, no door should remain open forever, but it will be eventually closed. However there is one thing that we want to positively avoid, and that is to harm any user who uses this system.

"Every time a door is opened, it will not remain in this state unless something is preventing it to close"

In order to check this property, we find it convenient to define three state predicates telling us when a door is opened, when is closed and when the sensors are warning us that something is in the door threshold. These state predicates are defined as follows:

```
eq < K | M | I | P | open | AC | L > |= dooropen = true .
eq < K | M | I | P | closed | AC | L > |= doorclosed = true .
eq < K | M | I | P | objectindoor | AC | L > |= protection = true .
```

Note that `dooropen`, `doorclosed` and `protection` are defined as operators of sort `Prop`. This way, the function in the `SATISFACTION` module, can be used to check if they are true or not. In this case, `dooropen` will be true when a door is opened, `doorclosed` will be true when a door is closed, and `protection` will be true when there is an object in the threshold.

Once we have the state predicates, we need to translate the statement of the property by using logical connectors. The LTL encoding is as follows:

```
[] ((~ dooropen) W <> (doorclosed \/  protection ))
```

and the `Maude` command to check the property, setting the initial state as `initial1`, is the following:

```
red modelCheck (initial1,[]((~ dooropen) W <>(doorclosed \/ protection))).
```

The result can be seen below and proves that the considered safety property is satisfied.

```
reduce in KEY2PHONE-CHECK in : modelCheck (initial1,[]((∼ dooropen) W
<>(doorclosed ∨ protection ))).
rewrites: 259 in 1ms (0 rewrites/second)
result Bool: true
```

"A door wont open unless a correct PIN code has been previously entered".

For assuring this conditional safety property, we need to check the opening of the doors and when a correct PIN is entered into the system. The state predicates for the opening of a door has already been declared, but we need a similar state predicate for the PIN. It is declared as follows:

```
eq < K | M | I | correctpin | D | AC | L > |= verifiedpin = true .
```

This predicate will be true every time a user types a correct PIN. The corresponding LTL encoding for this property is as follows:

```
[]((~dooropen) W verifiedpin)
```

Then, the Maude LTL model checker command that verifies this property is:

```
red modelCheck (initial1 ,[]((~ staynonprimary ) W [] staynonprimary)).
```

The result, proving that the property is met, can be seen below.

```
reduce in KEY2PHONE-CHECK in : modelCheck (initial1, [](∼ dooropen W verifiedpin)).
rewrites: 272 in 2ms (0 rewrites/second)
result Bool: true
```

"Always, in the event of any emergency signal, all doors are immediately opened"

avoiding any harm to the users is a vital part of this system. This means that besides from keeping the users from being harmed by a malfunction of the system, we also have to ensure that the users are safe when the danger comes from outside of the system (i.e., the building is on fire). In order to check this property, we need to check when an emergency alert is sent, as well as when the system open all the doors. This is quite easy because there are two actions specified in the *Emergency mode* to deal with these two events. The state predicates that we define are the following:

```
eq < K | M | I | P | D | notifyemergency | L > |= notify = true .
eq < K | M | I | P | D | opendoors | L > |= dooroverriden = true .
```

With the `notify` predicate, we are able to check when the notification is sent to all the users, then, when the doors open, the `dooroverriden` predicate is also satisfied.

Now, the LTL encoding for this property is specified as follows:

```
[] ((~ notify) W O dooroverriden)
```

Now we are able to check the properties by using the following `Maude` command, starting, as in the previous checks, from the state `initial1`:

```
red modelCheck (initial1, [] ((~ notify) W O dooroverriden)) .
```

We can see below that the property is satisfied in the system.

```
reduce in KEY2PHONE-CHECK in : modelCheck (initial1,[]((∼ notify) W O
dooroverriden )).
rewrites: 237 in 1ms (0 rewrites/second)
result Bool: true
```

"All the doors will only open simultaneously when the system is in the emergency mode"

It is true that giving the ability to open all the doors to the system in case of emergency is almost mandatory, but it also creates one of the biggest flaws of the system, which can be exploited by someone with malicious intentions.

We will use the `dooroverride` state predicate defined before, and add a new state predicate that warns us when the *Emergency mode* is entered.

```
eq < K | emergency | I | P | D | AC | L > |= emergencymode = true .
```

With this new state predicate we can define the LTL encoding as follows:

```
[] ((~ dooroverriden) W emergencymode)
```

The following `Maude` command is used to check if the property is true:

```
red modelCheck (initial1, [] ((~ dooroverriden) W emergencymode)) .
```

and the result, proving that the property is true, can be seen below.

```
reduce in KEY2PHONE-CHECK in : modelCheck (initial1,[](~ dooroverriden W
emergencymode ).
rewrites: 238 in 1ms (0 rewrites/second)
result Bool: true
```

"Any user logged using a Non-primary key will remain with this key unless he or she re-logs"

In order to check this property, the following equational predicate definition is added:

```
eq < nonprimary | M | I | P | D | AC | L > |= staynonprimary = true .
```

With this state predicate, `staynonprimary` will remain true as long as the `Key` keeps the `nonprimary` value. Then, the translated statement is:

```
[] ((~ staynonprimary ) W [] staynonprimary)
```

and we can use this `Maude` command to check the property in the system:

```
red modelCheck (initial1 , [] ((~ staynonprimary ) W [] staynonprimary) ) .
```

Below we can see the result of the command executed, verifying that the property holds.

```
reduce in KEY2PHONE-CHECK in : modelCheck (initial1,[](~ staynonprimary W
[] staynonprimary )).
rewrites: 287 in 1ms (0 rewrites/second)
result Bool: true
```

"When an account is blocked, the user wont be able to unlock it"

The state predicate defined to check this property is similar to the one defined for the previous property, but this time we monitor when the second parameter changes to `blocked`.

```
eq < K | blocked | I | P | D | AC | L > |= stayblocked = true .
```

Now we can translate the statement into LTL as follows:

```
[] ((~ stayblocked ) W [] stayblocked)
```

The corresponding `Maude` command used to check this property is:

```
red modelCheck (initial1, [] ((~ stayblocked ) W [] stayblocked))
```

and the result that verifies the property can be seen below.

```
reduce in KEY2PHONE-CHECK in : modelCheck (initial1,[](~ stayblocked W []
stayblocked )).
rewrites: 250 in 1ms (0 rewrites/second)
result Bool: true
```

"In every possible execution there will be a moment when the log will be updated"

We cannot assure that, in every possible execution, there will be a moment when a door opens. This is because, in the *Manage mode*, no action of opening a door will ever happen. But there is a thing that happens in every execution, the update of the system's log with the actions done during the execution. In order to prove this statement, we have to define a state predicate for every different form of update that has been defined in the system. The following equations are used:

```
eq < K | M | I | P | D | AC |
 updatelog(nouser, blocked) > |= logupdated = true .
eq < K | M | I | P | D | AC |
 updatelog(primary, blocked) > |= logupdated = true .
eq < K | M | I | P | D | AC |
 updatelog(nonprimary, blocked) > |= logupdated = true .
eq < K | M | I | P | D | AC |
 updatelog(primary, doorid) > |= logupdated = true .
eq < K | M | I | P | D | AC |
 updatelog(nonprimary, doorid) > |= logupdated = true .
eq < K | M | I | P | D | AC |
 updatelog(primary, adduser) > |= logupdated = true .
eq < K | M | I | P | D | AC |
 updatelog(primary, removeuser) > |= logupdated = true .
eq < K | M | I | P | D | AC |
 updatelog(primary, updateuser) > |= logupdated = true .
eq < K | M | I | P | D | AC |
 updatelog(primary, downloadreport) > |= logupdated = true .
eq < K | M | I | P | D | AC |
 updatelog(primary, configuresecurityrules) > |= logupdated = true .
eq < K | M | I | P | D | AC |
 updatelog(primary, notifyemergency) > |= logupdated = true .
eq < K | M | I | P | D | AC |
 updatelog(nonprimary, notifyemergency) > |= logupdated = true .
eq < K | M | I | P | D | AC |
 updatelog(primary, applysleepmodepolicy) > |= logupdated = true .
eq < K | M | I | P | D | AC |
 updatelog(primary, activateremotemodesecurity) > |= logupdated = true .
```

Every time the system updates the log, `logupdated` turns true; this way we can introduce the following LTL encoding of the statement:

```
[] <> logupdated
```

and the corresponding `Maude` command that proves this property is:

```
red modelCheck (initial1, [] <> logupdated ) .
```

The result of the execution can be seen below and proves that the last property holds as well.

```
reduce in KEY2PHONE-CHECK in : modelCheck (initial1,[] <> logupdated).
rewrites: 242 in 1ms (0 rewrites/second)
result Bool: true
```

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

In this master's thesis, we checked the safety properties of the Key2phone system by verifying that the safety requirements proposed in [CLB+15] are met. Moreover, additional liveness properties that prove the correct functioning of the modeled system have been also checked.

In order to accomplish this, first we defined a formal specification of the Key2phone system by using the Maude language. Then, we performed some reachability tests to ensure that every intended state is reached at some point and complemented this analysis by using the Anima tool [ABFS16], which helped us to visually check the computation trees generated by the Key2phone model. We also made use of the *Maude LTL Logical Model Checker* to express and verify some liveness properties that were beyond the scope of the previous reachability analyses and provide further guarantees of the correct functioning of the system.

The results of the analyses made in this thesis prove that the design of the Key2phone system is able to solve the vulnerabilities/threats ascertained in [CLB+15], this meaning that it correctly handles malicious users, and that the mechanism implemented to avoid any harm to a human being works properly.

As future work, we aim to achieve a more compact and optimized code. Also we plan to implement the system using *Full-Maude* in order to make a more thoughtful analysis of the system.

APPENDIX A. KEY2PHONE SPECIFICATION

```
load model-checker.maude .
mod KEY2PHONE is
protecting INT .


--- Sort Declaration
sorts PrimaryKey NonPrimaryKey Id Password Key Mode Status
Generalmode Adminmode Transitionrule Pin Pinmode Door
DoorId DoorStatus Log Actions Maction
BlockedMode NonBlockedMode .

--- Everything related to the Status needs to be put in this
subsort

subsort Transitionrule Key Mode Pin Door Log Actions < Status .
subsort PrimaryKey NonPrimaryKey < Key .
subsort Id Password < Key .
subsort Pinmode Adminmode Generalmode < NonBlockedMode < Mode .
subsort BlockedMode NonBlockedMode < Mode .
subsort DoorId DoorStatus < Door .
subsort Maction < Actions .

--- Initialize ops
ops admin noadmin nouser : -> Id .
ops password nopassword : -> Password .
ops counter : -> Int .
ops doorselection nodoor : -> Door .
ops nopin correctpin incorrectpin trypin : -> Pin .
op doorid : -> DoorId .
ops open protected closed objectindoor : -> DoorStatus .
op init : -> Status .
ops actionexpected applysleepmodepolicy protectdoorentrapment
notifyemergency activateremotemodesecurity stop opendoors
noaction : -> Actions .
op nolog : -> Log .



op primary : -> PrimaryKey .
op nonprimary : -> NonPrimaryKey .

ops logged unlogged : -> NonBlockedMode .
op blocked : -> BlockedMode .
ops management sleep remote : -> Adminmode .
op emergency : -> Generalmode .
ops normal move : -> Pinmode .

ops adduser removeuser updateuser downloadreport
configuresecurityrules : -> Maction .

--- Here, the sort Transitionrule is used
in order to keep clean the result when a search is done

op login : Key -> Transitionrule .

--- op changemode : Mode -> Transitionrule .

--- op checkpin : Pin -> Transitionrule .

---op dialnumber : Door -> Transitionrule .

op opendoor : Door -> Door .
op updatelog : Log -> Log .

--- op executeaction : Actions -> Actions .

--- op closedoor : Bool -> DoorStatus .

--- op <_|_|_|_> : Key Mode Int Pin -> Status .

op <_|_|_|_|_|_|_> : Key Mode Int Pin Door Actions Log
-> Status .

--- op <_|_|_|_|_> : Key Mode Int Pin Door -> Status .

--- op <_|_|_|_|_> : Key Mode Int Pin Actions -> Status .

--- op <_|_|_|_|_|_> : Key Mode Int Pin Actions Log -> Status .



op _,_ : Id Password -> PrimaryKey .

op _,_ : DoorId DoorStatus -> Door .

op _,_ : Key DoorId -> Log .

op _,_ : Key Actions -> Log .

op _,_ : Key Mode -> Log .



--- Declaration of the vars
var N : Int .
var KEY : Key .
var MODE : Mode .
var ADMINM : Adminmode .
var GENERALM : Generalmode .
var PINM : Pinmode .
var NBLOCKEDM : NonBlockedMode .
var DOORID : DoorId .
var MACTION : Maction .
var PIN : Pin .
var OIBD : Bool . --- OIBD stands for Object In Between Doors



--- Initialize status op
eq init = < nouser | unlogged | counter
| nopin | nodoor | noaction | nolog > .

eq counter = 2 .


--- Transitionrule nouser to login

rl < nouser | unlogged | N | nopin | nodoor | noaction
| nolog >
=>
< login( admin , password ) | unlogged | N | nopin
| nodoor
| noaction | nolog > .

rl < nouser | unlogged | N | nopin | nodoor | noaction
| nolog >
=>
< login( admin , nopassword ) | unlogged | N | nopin
| nodoor
| noaction | nolog > .

rl < nouser | unlogged | N | nopin | nodoor | noaction
| nolog >
=>
< login( noadmin ) | unlogged | N | nopin | nodoor
| noaction | nolog > .
```

```
--- Transitionrule logged to MODE


rl < primary | logged | N | nopin | nodoor | noaction
| nolog >
=>
< primary | management | N | nopin | nodoor | noaction
| nolog > .

rl < primary | logged | N | nopin | nodoor | noaction
| nolog >
=>
< primary | emergency | N | nopin | nodoor | noaction
| nolog > .

rl < primary | logged | N | nopin | nodoor | noaction
| nolog >
=>
< primary | move | N | nopin | nodoor | noaction
| nolog > .

rl < primary | logged | N | nopin | nodoor | noaction
| nolog >
=>
< primary | remote | N | nopin | nodoor | noaction
| nolog > .

rl < primary | logged | N | nopin | nodoor | noaction
| nolog >
=>
< primary | sleep | N | nopin | nodoor | noaction
| nolog > .

rl < primary | logged | N | nopin | nodoor | noaction
| nolog >
=>
< primary | normal | N | nopin | nodoor | noaction
| nolog > .
```

```
rl < nonprimary | logged | N | nopin | nodoor | noaction
| nolog >
=>
< nonprimary | emergency | N | nopin | nodoor | noaction
| nolog > .

rl < nonprimary | logged | N | nopin | nodoor | noaction
| nolog >
=>
< nonprimary | move | N | nopin | nodoor | noaction
| nolog > .

rl < nonprimary | logged | N | nopin | nodoor | noaction
| nolog >
=>
< nonprimary | normal | N | nopin | nodoor | noaction
| nolog > .

--- Transitionrule nopin to trypin

rl trypin => correctpin .
rl trypin => incorrectpin .


--- Transitionrule accessmode to dialnumber

rl doorselection => doorid .

--- Transitionrule actionexpected to MACTION

rl actionexpected => adduser .
rl actionexpected => removeuser .
rl actionexpected => updateuser .
rl actionexpected => downloadreport .
rl actionexpected => configuresecurityrules .

--- Transitionrule protected to doorstate
```

```
rl protected => closed .
rl protected => objectindoor .

--- Rules that change the Key

rl < login( admin , password ) | unlogged | N | nopin
| nodoor | noaction | nolog >
=>
< primary | logged | counter | nopin | nodoor | noaction
| nolog > .

crl < login ( admin , nopassword ) | unlogged | N | nopin
| nodoor | noaction | nolog >
=>
< nouser | unlogged | N - 1 | nopin | nodoor | noaction
| nolog >
if ( N > 0 ) .

rl < login ( admin , nopassword ) | unlogged | 0 | nopin
| nodoor | noaction | nolog >
=>
< nouser | blocked | 0 | nopin  | nodoor
| noaction | updatelog (nouser, blocked) > .

rl < login ( noadmin ) | unlogged | N | nopin | nodoor
| noaction | nolog >
=>
< nonprimary | logged | counter | nopin | nodoor
| noaction | nolog > .


--- Rules that change the mode

rl < primary | PINM | N | nopin | nodoor | noaction | nolog >
=>
< primary | PINM | N | trypin | nodoor | noaction | nolog > .
```

```
rl < nonprimary | PINM | N | nopin | nodoor | noaction
| nolog >
=>
< nonprimary | PINM | N | trypin | nodoor | noaction | nolog >.

rl < nonprimary | ADMINM | N | nopin | nodoor | noaction
| nolog >
=>
< nonprimary | logged | N | nopin | nodoor | noaction | nolog >.

--- Rules to check the pin

crl < KEY | PINM | N | correctpin | nodoor | noaction | nolog >
=>
< KEY | PINM | counter | correctpin | doorselection | noaction
| nolog > if ( N > 0 ) .

crl < KEY | PINM | N | incorrectpin | nodoor | noaction
| nolog >
=>
< KEY | PINM | N - 1 | trypin | nodoor | noaction | nolog >
if ( N > 0 ) .

--- Rules to update the log

rl < KEY | PINM | N | correctpin | doorid | noaction
| nolog >
=>
< KEY | PINM | counter | correctpin | opendoor ( doorid ) |
noaction | nolog  > .

rl < KEY | PINM | N | correctpin | opendoor ( doorid ) |
noaction | nolog >
=>
< KEY | PINM | counter | correctpin | open | noaction
| updatelog ( KEY , doorid ) > .

rl < KEY | PINM | N | correctpin | open | noaction
```

```
| updatelog ( KEY , doorid ) >
=>
< KEY | PINM | counter | correctpin | protected
| noaction | updatelog ( KEY , doorid ) > .


--- equations for the actions in management mode

rl < primary | management | N | nopin | nodoor | noaction
| nolog  >
=>
< primary | management | N | nopin | nodoor | actionexpected
| nolog  > .

rl < primary | management | N | nopin | nodoor | MACTION
| nolog >
=>
< primary | management | N | nopin | nodoor | MACTION
| updatelog ( primary , MACTION ) > .


--- equations for the actions in sleep mode

rl < primary | sleep | N | nopin | nodoor | noaction | nolog >
=> < primary | sleep | N | nopin | nodoor
| applysleepmodepolicy
| updatelog ( primary , applysleepmodepolicy) > .

rl < primary | sleep | N | nopin | nodoor
| applysleepmodepolicy
| updatelog ( primary , applysleepmodepolicy) >
=>
< primary | sleep | N | nopin | protected | noaction
| updatelog ( primary , applysleepmodepolicy) > .

--- equations for the actions in emergency mode

rl < KEY | emergency | N | nopin | nodoor | noaction | nolog >
```

```
=>
< KEY | emergency | N | nopin | nodoor | notifyemergency
| nolog > .

rl < KEY | emergency | N | nopin | nodoor | notifyemergency
| nolog >
=>
< KEY | emergency | N | nopin | nodoor | opendoors
| nolog > .

rl < KEY | emergency | N | nopin | nodoor | opendoors
| nolog >
=>
< KEY | emergency | N | nopin | nodoor | stop
| updatelog ( KEY, notifyemergency) > .

--- equations for the actions in remote mode

rl < primary | remote | N | nopin | nodoor | noaction | nolog >
=>
< primary | remote | N | nopin | nodoor
| activateremotemodesecurity
| updatelog ( primary, activateremotemodesecurity ) > .

rl < primary | remote | N | nopin | nodoor
| activateremotemodesecurity
| updatelog ( primary, activateremotemodesecurity ) >
=>
< primary | remote | N | nopin | protected | noaction
| updatelog ( primary, activateremotemodesecurity ) > .


endm

mod KEY2PHONE-PREDS is
protecting KEY2PHONE .
including SATISFACTION .
```

```
subsort State < Status .

ops verifiedpin dooropen staynonprimary emergencymode
notify doorclosed protection dooroverriden stayblocked
logupdated : -> Prop .

var K : Key .
var M : Mode .
var I : Int .
var P : Pin .
var D : Door .
var AC : Actions .
var L : Log .
var S : Status .
var PR : Prop .


---emergency state predicates

eq < K | emergency | I | P | D | AC | L >
|= emergencymode = true .
eq < K | M | I | P | D | notifyemergency | L >
|= notify = true .
eq < K | M | I | P | D | opendoors | L >
|= dooroverriden = true .
eq < K | M | I | correctpin | D | AC | L >
|= verifiedpin = true .

---door status state predicates

eq < K | M | I | P | open | AC | L >
|= dooropen = true .
eq < K | M | I | P | closed | AC | L >
|= doorclosed = true .
eq < K | M | I | P | objectindoor | AC | L >
|= protection = true .

eq < nonprimary | M | I | P | D | AC | L >
```

```
|= staynonprimary = true .
eq < K | blocked | I | P | D | AC | L >
|= stayblocked = true .


--- Log state predicates

eq < K | M | I | P | D | AC | updatelog(nouser, blocked) >
|= logupdated = true .

eq < K | M | I | P | D | AC | updatelog(primary, blocked) >
|= logupdated = true .

eq < K | M | I | P | D | AC | updatelog(nonprimary, blocked) >
|= logupdated = true .

eq < K | M | I | P | D | AC | updatelog(primary, doorid) >
|= logupdated = true .

eq < K | M | I | P | D | AC | updatelog(nonprimary, doorid) >
|= logupdated = true .

eq < K | M | I | P | D | AC | updatelog(primary, adduser) >
|= logupdated = true .

eq < K | M | I | P | D | AC | updatelog(primary, removeuser) >
|= logupdated = true .

eq < K | M | I | P | D | AC | updatelog(primary, updateuser) >
|= logupdated = true .

eq < K | M | I | P | D | AC
| updatelog(primary, downloadreport) > |= logupdated = true .

eq < K | M | I | P | D | AC
| updatelog(primary, configuresecurityrules) >
|= logupdated = true .
```

```
eq < K | M | I | P | D | AC
| updatelog(primary, notifyemergency) >
|= logupdated = true .

eq < K | M | I | P | D | AC
| updatelog(nonprimary, notifyemergency) >
|= logupdated = true .

eq < K | M | I | P | D | AC
| updatelog(primary, applysleepmodepolicy) >
|= logupdated = true .

eq < K | M | I | P | D | AC
| updatelog(primary, activateremotemodesecurity) >
|= logupdated = true .

endm

mod KEY2PHONE-CHECK is
protecting KEY2PHONE-PREDS .
including MODEL-CHECKER .
including LTL-SIMPLIFIER .

ops initial1 initial2 initial3 initial4 : -> Status .

eq initial1 = < nouser | unlogged | counter | nopin | nodoor
| noaction | nolog > .
endm
```

APPENDIX B. PROPERTIES SPECIFICATION

```
***(

This file is part of the Maude 2 interpreter.

Copyright 1997-2006 SRI International,
Menlo Park, CA 94025, USA.

This program is free software; you can redistribute it and/or
modify it under the terms of the GNU General Public License
as published by the Free Software Foundation; either
version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General
Public License along with this program; if not,
write to the Free Software Foundation, Inc., 59
Temple Place, Suite 330, Boston, MA 02111-1307, USA.

)
```

```
***
*** Maude LTL satisfiability solver and model checker.
*** Version 2.3.
***

fmod LTL is
protecting BOOL .
sort Formula .

*** primitive LTL operators
ops True False : -> Formula [ctor format (g o)] .
op ~_ : Formula -> Formula [ctor prec 53 format (r o d)] .
op _/\_ : Formula Formula -> Formula [comm ctor gather (E e)
prec 55 format (d r o d)] .
op _\/_ : Formula Formula -> Formula [comm ctor gather
(E e) prec 59 format (d r o d)] .
op O_ : Formula -> Formula [ctor prec 53 format (r o d)] .
op _U_ : Formula Formula ->
Formula [ctor prec 63 format (d r o d)] .
op _R_ : Formula Formula ->
Formula [ctor prec 63 format (d r o d)] .

*** defined LTL operators
op _->_ : Formula Formula ->
Formula [gather (e E) prec 65 format (d r o d)] .
op _<->_ : Formula Formula ->
Formula [prec 65 format (d r o d)] .
op <>_ : Formula ->
Formula [prec 53 format (r o d)] .
op []_ : Formula ->
Formula [prec 53 format (r d o d)] .
op _W_ : Formula Formula ->
Formula [prec 63 format (d r o d)] .
op _|->_ : Formula Formula ->
Formula [prec 63 format (d r o d)] . *** leads-to
op _=>_ : Formula Formula ->
Formula [gather (e E) prec 65 format (d r o d)] .
op _<=>_ : Formula Formula ->
```

```
Formula [prec 65 format (d r o d)] .

vars f g : Formula .

eq f -> g = ~ f \/ g .
eq f <-> g = (f -> g) /\ (g -> f) .
eq <> f = True U f .
eq [] f = False R f .
eq f W g = (f U g) \/ [] f .
eq f |-> g = [](f -> (<> g)) .
eq f => g = [] (f -> g) .
eq f <=> g = [] (f <-> g) .

*** negative normal form
eq ~ True = False .
eq ~ False = True .
eq ~ ~ f = f .
eq ~ (f \/ g) = ~ f /\ ~ g .
eq ~ (f /\ g) = ~ f \/ ~ g .
eq ~ O f = O ~ f .
eq ~(f U g) = (~ f) R (~ g) .
eq ~(f R g) = (~ f) U (~ g) .
endfm

fmod LTL-SIMPLIFIER is
including LTL .

*** The simplifier is based on:
***   Kousha Etessami and Gerard J. Holzman,
***   "Optimizing Buchi Automata",
p153-167, CONCUR 2000, LNCS 1877.
*** We use the Maude sort system to do much of the work.

sorts TrueFormula FalseFormula PureFormula
PE-Formula PU-Formula .
subsort TrueFormula FalseFormula < PureFormula <
PE-Formula PU-Formula < Formula .
```

```
op True : -> TrueFormula [ctor ditto] .
op False : -> FalseFormula [ctor ditto] .
op _/\_ : PE-Formula PE-Formula -> PE-Formula [ctor ditto] .
op _/\_ : PU-Formula PU-Formula -> PU-Formula [ctor ditto] .
op _/\_ : PureFormula PureFormula -> PureFormula [ctor ditto] .
op _\/_ : PE-Formula PE-Formula -> PE-Formula [ctor ditto] .
op _\/_ : PU-Formula PU-Formula -> PU-Formula [ctor ditto] .
op _\/_ : PureFormula PureFormula -> PureFormula [ctor ditto] .
op O_ : PE-Formula -> PE-Formula [ctor ditto] .
op O_ : PU-Formula -> PU-Formula [ctor ditto] .
op O_ : PureFormula -> PureFormula [ctor ditto] .
op _U_ : PE-Formula PE-Formula -> PE-Formula [ctor ditto] .
op _U_ : PU-Formula PU-Formula -> PU-Formula [ctor ditto] .
op _U_ : PureFormula PureFormula -> PureFormula [ctor ditto] .
op _U_ : TrueFormula Formula -> PE-Formula [ctor ditto] .
op _U_ : TrueFormula PU-Formula -> PureFormula [ctor ditto] .
op _R_ : PE-Formula PE-Formula -> PE-Formula [ctor ditto] .
op _R_ : PU-Formula PU-Formula -> PU-Formula [ctor ditto] .
op _R_ : PureFormula PureFormula -> PureFormula [ctor ditto] .
op _R_ : FalseFormula Formula -> PU-Formula [ctor ditto] .
op _R_ : FalseFormula PE-Formula -> PureFormula [ctor ditto] .

vars p q r s : Formula .
var pe : PE-Formula .
var pu : PU-Formula .
var pr : PureFormula .

*** Rules 1, 2 and 3; each with its dual.
eq (p U r) /\ (q U r) = (p /\ q) U r .
eq (p R r) \/ (q R r) = (p \/ q) R r .
eq (p U q) \/ (p U r) = p U (q \/ r) .
eq (p R q) /\ (p R r) = p R (q /\ r) .
eq True U (p U q) = True U q .
eq False R (p R q) = False R q .

*** Rules 4 and 5 do most of the work.
eq p U pe = pe .
eq p R pu = pu .
```

```
*** An extra rule in the same style.
eq O pr = pr .

*** We also use the rules from:
***   Fabio Somenzi and Roderick Bloem,
*** "Efficient Buchi Automata from LTL Formulae",
***   p247-263, CAV 2000, LNCS 1633.
*** that are not subsumed by the previous system.

*** Four pairs of duals.
eq O p /\ O q = O (p /\ q) .
eq O p \/ O q = O (p \/ q) .
eq O p U O q = O (p U q) .
eq O p R O q = O (p R q) .
eq True U O p = O (True U p) .
eq False R O p = O (False R p) .
eq (False R (True U p)) \/ (False R (True U q)) =
False R (True U (p \/ q)) .
eq (True U (False R p)) /\ (True U (False R q)) =
True U (False R (p /\ q)) .

*** <= relation on formula
op _<=_ : Formula Formula -> Bool [prec 75] .

eq p <= p = true .
eq False <= p  = true .
eq p <= True = true .

ceq p <= (q /\ r) = true if (p <= q) /\ (p <= r) .
ceq p <= (q \/ r) = true if p <= q .
ceq (p /\ q) <= r = true if p <= r .
ceq (p \/ q) <= r = true if (p <= r) /\ (q <= r) .

ceq p <= (q U r) = true if p <= r .
ceq (p R q) <= r = true if q <= r .
ceq (p U q) <= r = true if (p <= r) /\ (q <= r) .
ceq p <= (q R r) = true if (p <= q) /\ (p <= r) .
ceq (p U q) <= (r U s) = true if (p <= r) /\ (q <= s) .
ceq (p R q) <= (r R s) = true if (p <= r) /\ (q <= s) .

*** condition rules depending on <= relation
ceq p /\ q = p if p <= q .
ceq p \/ q = q if p <= q .
ceq p /\ q = False if p <= ~ q .
ceq p \/ q = True if ~ p <= q .
ceq p U q = q if p <= q .
ceq p R q = q if q <= p .
ceq p U q = True U q if p =/= True /\ ~ q <= p .
ceq p R q = False R q if p =/= False /\ q <= ~ p .
ceq p U (q U r) = q U r if p <= q .
ceq p R (q R r) = q R r if q <= p .
endfm

fmod SAT-SOLVER is
protecting LTL .

*** formula lists and results
sorts FormulaList SatSolveResult TautCheckResult .
subsort Formula < FormulaList .
subsort Bool < SatSolveResult TautCheckResult .
op nil : -> FormulaList [ctor] .
op _;_ : FormulaList FormulaList ->
FormulaList [ctor assoc id: nil] .
op model : FormulaList FormulaList -> SatSolveResult [ctor] .

op satSolve : Formula ~> SatSolveResult
[special (
id-hook SatSolverSymbol
op-hook trueSymbol           (True : ~> Formula)
op-hook falseSymbol (False : ~> Formula)
op-hook notSymbol (~_ : Formula ~> Formula)
op-hook nextSymbol (O_ : Formula ~> Formula)
op-hook andSymbol (_/\_ : Formula Formula ~> Formula)
op-hook orSymbol (_\/_ : Formula Formula ~> Formula)
op-hook untilSymbol (_U_ : Formula Formula ~> Formula)
op-hook releaseSymbol (_R_ : Formula Formula ~> Formula)
op-hook formulaListSymbol
(_;_ : FormulaList FormulaList ~> FormulaList)
op-hook nilFormulaListSymbol (nil : ~> FormulaList)
op-hook modelSymbol
(model : FormulaList FormulaList ~> SatSolveResult)
term-hook falseTerm (false)
)] .

op counterexample : FormulaList FormulaList ->
TautCheckResult [ctor] .
op tautCheck : Formula ~> TautCheckResult .
op $invert : SatSolveResult -> TautCheckResult .

var F : Formula .
vars L C : FormulaList .
eq tautCheck(F) = $invert(satSolve(~ F)) .
eq $invert(false) = true .
eq $invert(model(L, C)) = counterexample(L, C) .
endfm

fmod SATISFACTION is
protecting BOOL .
sorts State Prop .
op _|=_ : State Prop -> Bool [frozen] .
endfm

fmod MODEL-CHECKER is
protecting QID .
including SATISFACTION .
including LTL .
subsort Prop < Formula .

*** transitions and results
sorts RuleName Transition TransitionList ModelCheckResult .
subsort Qid < RuleName .
subsort Transition < TransitionList .
subsort Bool < ModelCheckResult .
```

```
ops unlabeled deadlock : -> RuleName .
op {_,_} : State RuleName -> Transition [ctor] .
op nil : -> TransitionList [ctor] .
op __ : TransitionList TransitionList ->
TransitionList [ctor assoc id: nil] .
op counterexample : TransitionList TransitionList ->
ModelCheckResult [ctor] .

op modelCheck : State Formula ~> ModelCheckResult
[special (
id-hook ModelCheckerSymbol
op-hook trueSymbol           (True : ~> Formula)
op-hook falseSymbol (False : ~> Formula)
op-hook notSymbol (~_ : Formula ~> Formula)
op-hook nextSymbol (O_ : Formula ~> Formula)
op-hook andSymbol (_/\_ : Formula Formula ~> Formula)
op-hook orSymbol (_\/_ : Formula Formula ~> Formula)
op-hook untilSymbol (_U_ : Formula Formula ~> Formula)
op-hook releaseSymbol (_R_ : Formula Formula ~> Formula)
op-hook satisfiesSymbol      (_|=_ : State Formula ~> Bool)
op-hook qidSymbol (<Qids> : ~> Qid)
op-hook unlabeledSymbol (unlabeled : ~> RuleName)
op-hook deadlockSymbol (deadlock : ~> RuleName)
op-hook transitionSymbol ({_,_} : State RuleName ~> Transition)
op-hook transitionListSymbol
(__ : TransitionList TransitionList ~> TransitionList)
op-hook nilTransitionListSymbol (nil : ~> TransitionList)
op-hook counterexampleSymbol
(counterexample :
TransitionList TransitionList ~> ModelCheckResult)
term-hook trueTerm (true)
)] .
endfm}
```

BIBLIOGRAPHY

[ABFS15] M. Alpuente, D. Ballis, F. Frechina, and J. Sapiña. Exploring Conditional Rewriting Logic Computations. *Journal of Symbolic Computation*, 69:3–39, 2015.

[ABFS16] M. Alpuente, D. Ballis, F. Frechina, and J. Sapiña. Debugging Maude Programs via Runtime Assertion Checking and Trace Slicing. *Journal of Logical and Algebraic Methods in Programming*, 85:707–736, 2016.

[CDE$^+$16] M. Clavel, F. Durán, S. Eker, S. Escobar, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. Maude Manual (Version 2.7.1). Technical report, SRI Int'l Computer Science Lab., 2016.

[CLB$^+$15] S. Chaudhary, L. Li, E. Berki, M. Helenius, J. Kela, and M. Turunen. Applying Finite State Process Algebra to Formally Specify a Computational Model of Security Requeriments in the Key2phone-Mobile Access solution. *FMICS*, pages 128–145, 2015.

[CS10] Michael R. Clarkson and Fred B. Schneider. Hyperproperties. *Journal of Computer Security*, pages 1157–1210, 2010.

[DW97] J. Kelsey D. Wagner, B. Schneier. Crypanalysis of the Celluler Message Encryption Algorithm. In *CRYPTO'97*, pages 526–537, 1997.

[KC12] Conor Mc Garvey Kevin Curran, Amanda Millar. Near Field Communication. *International journal of electrical and computer engineering*, 2(3):371 382, 2012.

[Mes92] J. Meseguer. Conditional Rewriting Logic as a Unified Model of Concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[SS03] J. Meseguer S.Eker and A. Sridharanarayanan. The Maude LTL Model Checker. In *Proc. of the 8th Int'l Joint Conference on Automated Reasoning (IJCAR 2016)*, volume 71 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 2003.

[TeR03] TeReSe. *Term Rewriting Systems*. Cambridge University Press, 2003.