Especificación Formal de un Sistema de Control de Acceso para Laboratorios en la Universidad La Salle de Arequipa

Jhordan Huamaní Huamaní[1], Jorge Ortiz Castañeda[1], José Mamani Zuñiga[1], Miguel Flores León[1]
[1]Departamento de Ingeniería de Software, Universidad La Salle de Arequipa, Perú
{jhordanhh, jortizc, jmamamniz, mfloresl}@ulasalle.edu.pe

***Resumen*—Este artículo presenta el desarrollo del proyecto *Especificación Formal de un Sistema de Control de Acceso para Laboratorios en la Universidad La Salle de Arequipa*, realizado por estudiantes de la Universidad La Salle y expuesto en la Feria de Proyectos *La Salle In-Genio* 2025. El objetivo principal es diseñar la especificación formal del sistema (SICA-L) utilizando VDM++, abordando la vulnerabilidad de los equipos de alto valor y la actual falta de trazabilidad. La propuesta consiste en validar el modelo mediante herramientas como VDM++ Toolbox y NuSMV, destacando su contribución en términos de seguridad preventiva y la generación de un diseño libre de ambigüedades. Finalmente, se discuten los resultados preliminares y el potencial de aplicación en futuras implementaciones para instituciones de educación superior.**

***Index Terms*—Especificación formal, Modelos de control de acceso, Sistema de control de acceso, Verificación de modelos**

I. Introducción

La Universidad La Salle de Arequipa ha emprendido un proceso de modernización de su infraestructura tecnológica y académica, incorporando laboratorios con equipamiento de alto valor. En este contexto, surge la necesidad crítica de implementar un sistema que permita controlar, registrar y auditar los accesos a dichos espacios de manera formal y verificable. La gestión eficiente de estos recursos exige soluciones que garanticen no solo la seguridad física, sino también la trazabilidad inequívoca de las operaciones realizadas por el personal autorizado.

Los sistemas tradicionales de control, basados frecuentemente en registros manuales o mecanismos informales, resultan insuficientes ante los estándares actuales de seguridad. Esta carencia evidencia la vulnerabilidad ante errores humanos y dificulta la auditoría forense en caso de incidentes. Por consiguiente, resulta imperativo diseñar un mecanismo automatizado que incorpore criterios formales para eliminar ambigüedades, inconsistencias lógicas y brechas de seguridad que podrían comprometer la integridad institucional.

El presente artículo introduce el proyecto *Especificación Formal de un Sistema de Control de Acceso para Laboratorios (SICA-L)*, desarrollado por estudiantes de la Universidad La Salle y expuesto en la Feria de Proyectos *La Salle In-Genio 2025*. La propuesta central consiste en la aplicación de métodos formales, específicamente el lenguaje VDM++, para modelar las operaciones críticas del sistema tales como autenticación, autorización y registro asegurando la corrección matemática del diseño antes de su codificación.

Más allá de la especificación, el modelo es sometido a una rigurosa validación y verificación: se emplea análisis de cobertura para la lógica interna y herramientas de *model checking* como NuSMV y UPPAAL para garantizar el cumplimiento de propiedades de seguridad y restricciones de tiempo real. De este modo, el trabajo establece un *blueprint* formal y robusto, sirviendo como referencia replicable para futuras implementaciones en entornos académicos y de alta seguridad.

Presentado en la Feria de Proyectos *Inspira La Salle*, Universidad La Salle, Arequipa, 2025

II. Trabajos anteriores

La aplicación de métodos formales para la especificación y verificación de sistemas de control de acceso es un campo de investigación bien establecido, ya que la seguridad y la fiabilidad son requisitos no negociables en estos sistemas. El presente trabajo se sitúa en la intersección de varias áreas de investigación consolidadas.

El uso del *Vienna Development Method* (VDM) y su extensión orientada a objetos, VDM++, para modelar sistemas críticos tiene una larga trayectoria. Autores como Bryans y Fitzgerald han demostrado cómo VDM++ puede ser utilizado para la ingeniería formal de políticas de control de acceso complejas, como las expresadas en XACML, permitiendo un análisis riguroso antes de la implementación [1]. La versatilidad de VDM++ también se ha demostrado en la especificación de sistemas industriales, como los sistemas de autodefensa para aeronaves de combate, donde la precisión del modelo es fundamental para garantizar la seguridad operacional [2]. Además, su aplicación se extiende a sistemas de transacciones seguras, como monederos electrónicos, donde la especificación formal ha sido clave para detectar errores sutiles en el código fuente [3].

El *model checking* es la técnica de verificación por excelencia para encontrar fallos en sistemas concurrentes y de seguridad [4]. Herramientas como NuSMV y SPIN se han utilizado extensamente para analizar protocolos de seguridad y encontrar vulnerabilidades que las pruebas manuales no logran detectar, como en el famoso caso del protocolo Needham-Schroeder [5]. Se ha empleado, por ejemplo, para validar políticas de control de acceso expresadas en SecureUML, traduciéndolas a un lenguaje formal como B para realizar pruebas

sistemáticas de permisos de roles [6]. La formalización de propiedades de seguridad como autorización, autenticación, integridad y confidencialidad utilizando VDM-SL, el precursor de VDM++, ha demostrado la capacidad de estos métodos para especificar sistemas sin ambigüedades [7].

En el ámbito de los sistemas de tiempo real, donde las restricciones temporales son tan importantes como la corrección lógica, UPPAAL es la herramienta de referencia [17]. Se ha aplicado con éxito para verificar sistemas críticos como los controladores de semáforos inteligentes, donde un fallo en la temporización puede tener consecuencias catastróficas [9]. Su capacidad para modelar y verificar protocolos con restricciones de tiempo también lo hace adecuado para analizar aspectos temporales en las políticas de control de acceso, como las definidas en el modelo *Temporal Role-Based Access Control* (TRBAC) [10].

Este trabajo se distingue por su enfoque metodológico integral: no solo se especifica el sistema de control de acceso en VDM++, sino que se valida su lógica interna con análisis de cobertura y, crucialmente, se verifica formalmente con un doble enfoque de *model checking*. Se utiliza NuSMV para probar la robustez de las políticas de seguridad desde una perspectiva lógica y atemporal, y UPPAAL para garantizar que el comportamiento del sistema cumple con las restricciones de tiempo del mundo real. Esta combinación de técnicas proporciona un nivel de confianza en la correctitud del sistema que es difícil de alcanzar con un único método.

III. Objetivos

III-A. Objetivo General

Elaborar la especificación formal de un sistema de control de acceso para los nuevos laboratorios de la Universidad La Salle, que garantice la seguridad y la trazabilidad desde su puesta en marcha, y que sirva como modelo base escalable para otras instituciones de educación superior.

III-B. Objetivos Específicos

1. Modelar las entidades clave del sistema: Usuarios, Laboratorios y Permisos de acceso.
2. Especificar las políticas y reglas de acceso mediante un método formal para lograr la precisión necesaria en el diseño.
3. Definir las operaciones críticas (como la verificación de acceso y gestión de permisos) y establecer los invariantes del sistema para asegurar su consistencia lógica.

IV. Marco teórico

IV-A. Definición de Verificación Formal

La verificación formal es una disciplina de la ingeniería de software y hardware que utiliza métodos matemáticos para demostrar que un sistema cumple con un conjunto de propiedades o especificaciones formales [11]. A diferencia de las pruebas (*testing*), que solo pueden encontrar errores mediante la ejecución de un subconjunto de casos, la verificación formal tiene como objetivo probar la ausencia de ciertos tipos de errores en todos los comportamientos posibles del sistema [12].

Este proceso requiere tres componentes clave: un lenguaje formal con una semántica matemática precisa para describir el sistema, una especificación de las propiedades que el sistema debe cumplir, y un método de verificación (como el *model checking* o la demostración de teoremas) para probar que la descripción del sistema satisface la especificación [11].

IV-B. VDM++

IV-B1. Definición: VDM++ es un lenguaje de especificación formal orientado a objetos y basado en modelos, que evolucionó a partir de VDM-SL (*Specification Language*). Es especialmente adecuado para modelar sistemas complejos y con estado, describiendo el sistema en términos de los tipos de datos que maneja y las operaciones que modifican su estado [13].

IV-B2. Clases: Un modelo en VDM++ se estructura como un conjunto de clases, que encapsulan un estado interno (variables de instancia) y el comportamiento que opera sobre ese estado (operaciones y funciones) [13].

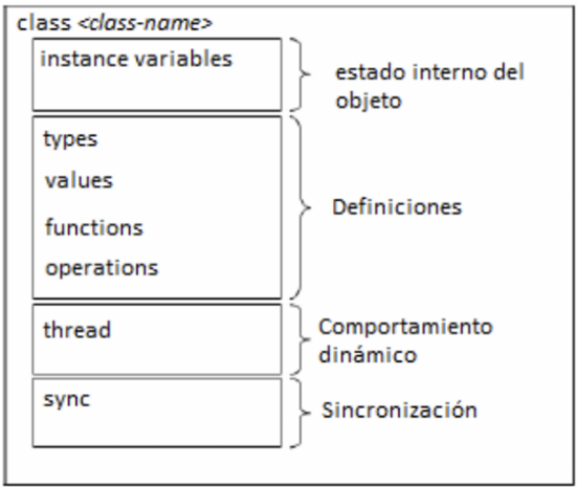

Figura 1: Estructura de una clase en VDM++.

IV-B3. Tipos: VDM++ posee un sistema de tipos robusto, que incluye tipos básicos como `nat` (números naturales), `bool` (booleanos) e `int` (enteros), así como tipos compuestos potentes como `set of` (conjuntos), `seq of` (secuencias) y `map to` (mapeos), que son fundamentales para modelar estructuras de datos complejas [14].

IV-B4. Invariantes: Una invariante de clase es una condición o predicado lógico sobre las variables de instancia que debe ser verdadero en todo momento (excepto durante la ejecución de una operación). Las invariantes son cruciales para definir la consistencia y la integridad de los datos del modelo [13].

```
class <nombre-clase>
  instance variables
  definición 1;
  definición 2;
  ...
  definición n;
  inv expresión utilizando las variables de instancia

end <nombre-clase>
```

Figura 2: Especificación de una invariante.

IV-B5. Pre y Postcondiciones: Las operaciones pueden ser especificadas mediante pre y postcondiciones. Una precondición es un predicado que debe cumplirse antes de que una operación pueda ser invocada. Una postcondición es un predicado que la operación garantiza que se cumplirá al finalizar, relacionando el estado final con el estado inicial. Juntas, forman un çontrato"que define el comportamiento de la operación [13].

IV-B6. Funciones y Operaciones: VDM++ distingue entre *operations*, que pueden modificar el estado de una clase (tienen efectos secundarios), y *functions*, que son puramente computacionales y no pueden cambiar el estado [13].

```
class <nombre-clase>
  functions
  nombreFunción: tipo parámetros +> tipo retorno
  nombreFunción (parámetros)==
     sentencias
  nombreInvariante: tipo parámetros +> bool
  nombreInvariante (parámetros)
     sentencias
end <nombre-clase>
```

Figura 3: Especificación de una función y una función invariante.

```
class <nombre-clase>
  operations
  nombreOperación: tipo parámetros ==> tipo retorno
  nombreOperación (parámetros)==
         sentencias
  pre expresión
  post expresión

end <nombre-clase>
```

Figura 4: Especificación de operaciones.

IV-B7. Herramienta (VDM++ Toolbox): Es un Entorno de Desarrollo Integrado (IDE) para VDM++. Proporciona herramientas para el análisis sintáctico y de tipos, un intérprete para ejecutar y depurar la especificación, un generador de pruebas y una herramienta de análisis de cobertura de código, que es esencial para la validación del modelo [13].

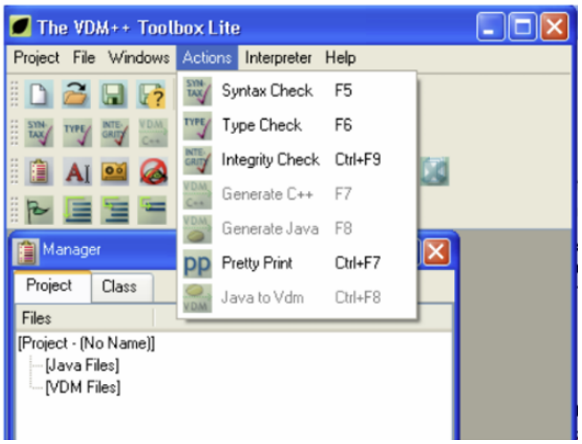

Figura 5: Interface de VDM++ Toolbox.

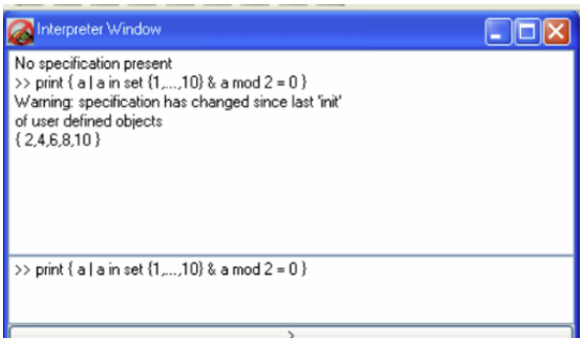

Figura 6: Interprete de VDM++ Toolbox.

IV-C. Model Checking

El *Model Checking* es una técnica de verificación formal automatizada que explora sistemáticamente todos los estados posibles de un sistema (su espacio de estados) para determinar si cumple una propiedad dada [12]. Las propiedades se expresan formalmente utilizando lógicas temporales, como la Lógica de Árbol Computacional (CTL) o la Lógica de Tiempo Lineal (LTL). Si el sistema no cumple una propiedad, la herramienta genera un contraejemplo: una traza de ejecución que demuestra cómo se viola la propiedad [11].

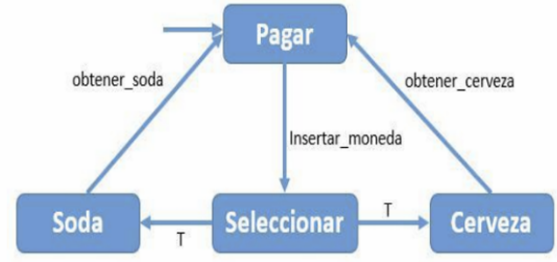

Figura 7: Ejemplo Model Checking.

IV-C1. NuSMV: Es un *symbolic model checker* que utiliza estructuras de datos eficientes como los Diagramas de Decisión Binarios (BDDs) para representar el espacio de estados

de forma compacta, lo que le permite analizar sistemas muy grandes [15]. Es ideal para verificar propiedades lógicas y de seguridad (*safety properties*, como ün estado inseguro nunca es alcanzable") y de vivacidad (*liveness properties*, como üna petición siempre será atendida eventualmente") sobre sistemas que no tienen un componente de tiempo real explícito.

IV-C2. UPPAAL: Es una caja de herramientas integrada para la modelización, simulación y verificación de sistemas de tiempo real. El modelo de un sistema en UPPAAL es una red de autómatas temporizados, que son autómatas de estados finitos extendidos con variables de reloj de valor real [16]. Las transiciones pueden tener guardas (condiciones sobre los relojes que deben cumplirse para que la transición ocurra) y los estados pueden tener invariantes (condiciones sobre los relojes que deben cumplirse para permanecer en el estado). UPPAAL es ideal para verificar propiedades que incluyen restricciones de tiempo [17].

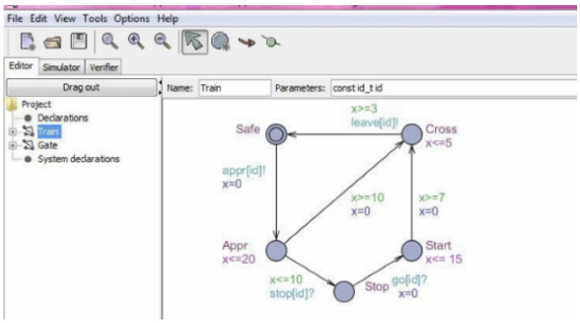

Figura 8: Ejemplo de UPPAAL, sistema modelado con 5 estados.

V. Metodología

La metodología empleada en este trabajo es de naturaleza computacional y formal, basada en el ciclo de vida de desarrollo de software riguroso para sistemas críticos. El proceso se centra en la transformación sistemática de requerimientos informales en especificaciones matemáticas precisas, permitiendo la detección temprana de ambigüedades y errores lógicos.

El desarrollo del proyecto se ha estructurado en cuatro etapas secuenciales e iterativas, tal como se ilustra en el diagrama de flujo de la Fig. 9, y se detalla a continuación:

V-A. Análisis y Abstracción de Requerimientos

En esta etapa inicial se realizó la recopilación de datos mediante la observación de los procesos actuales de acceso en los laboratorios de la Universidad La Salle. Se identificaron los actores (estudiantes, docentes, administrativos), los recursos (laboratorios) y las restricciones operativas.

- **Entrada:** Reglas de negocio en lenguaje natural y entrevistas con los responsables de laboratorio.
- **Salida:** Lista refinada de Requerimientos Funcionales (RF) y un Diagrama de Clases UML preliminar.

V-B. Especificación Formal con VDM++

Esta fase constituye el núcleo del modelado teórico. Se procedió a la formalización de la estructura estática y dinámica del sistema utilizando el lenguaje *Vienna Development Method* orientado a objetos (VDM++). Se definieron tres componentes matemáticos clave:

1. **Tipos de Datos:** Abstracción de entidades mediante `types` personalizados.
2. **Invariantes de Estado:** Ecuaciones lógicas que restringen los valores permitidos de las variables de instancia para mantener la integridad del sistema.

V-C. Validación Interna y Análisis de Cobertura

Para garantizar la consistencia interna del modelo, se empleó una metodología experimental de pruebas basadas en escenarios. Utilizando la herramienta *VDM++ Toolbox*.

El criterio de éxito establecido fue alcanzar un 100 % de cobertura de código (*statement coverage*), asegurando que todas las líneas de la especificación, incluyendo las cláusulas de manejo de excepciones, fueran ejercitadas durante la simulación.

V-D. Verificación de Modelos (Model Checking)

Finalmente, se realizó la verificación externa de propiedades críticas mediante técnicas de *Model Checking*, dividiendo el análisis en dos dominios complementarios:

V-D1. Seguridad Lógica (NuSMV): Se tradujo el modelo a una máquina de estados finitos para verificar propiedades de seguridad (*safety properties*) expresadas en Lógica de Árbol Computacional (CTL), asegurando matemáticamente que nunca ocurra un estado prohibido.

V-D2. Tiempo Real (UPPAAL): Se modeló el sistema como una red de autómatas temporizados para validar restricciones temporales (*liveness properties*), verificando que el sistema responda a las solicitudes de acceso dentro de los plazos establecidos (latencia máxima permitida).

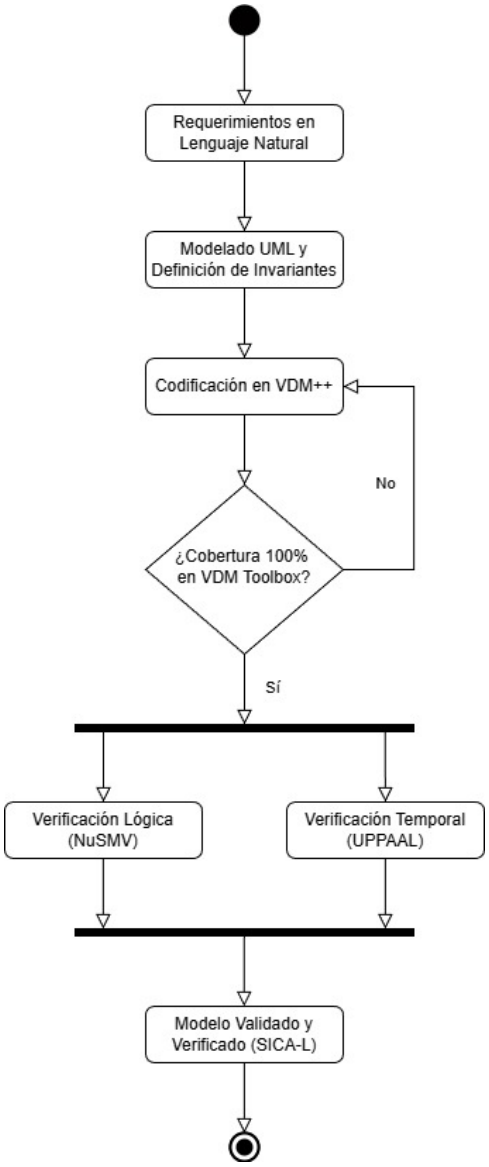

Figura 9: Diagrama de flujo de la metodología

VI. Desarrollo

En esta sección se detalla la aplicación práctica de la metodología formal al diseño y verificación del Sistema de Control de Acceso para Laboratorios (SICA-L).

VI-A. Planteamiento del problema

La Universidad La Salle de Arequipa se encuentra en un proceso de expansión de su infraestructura académica, dotando a nuevos laboratorios con equipos de alto valor, tecnología de punta y materiales de investigación sensibles. En este contexto, la ausencia de un sistema de control de acceso formal y robusto constituye una vulnerabilidad crítica. Los métodos tradicionales, como las bitácoras manuales, son insuficientes para este entorno, ya que son propensos a errores humanos, falsificación y carecen de capacidades de auditoría en tiempo real.

La falta de un sistema adecuado genera riesgos directos y significativos:

- **Riesgo de Seguridad:** Facilita el posible robo, daño o mal uso de equipos costosos, comprometiendo la inversión institucional.
- **Riesgo de Integridad Académica:** Permite la manipulación no autorizada de experimentos o datos de investigación.
- **Falta de Trazabilidad:** Impide determinar con certeza quién se encontraba en las instalaciones durante un incidente, dificultando la rendición de cuentas.

Dado que el sistema aún no existe, la universidad tiene la oportunidad única de diseñar e implementar una solución correcta desde su concepción, un enfoque conocido como *greenfield*.

VI-B. Solución propuesta

Se propone el diseño de un Sistema de Control de Acceso (SICA-L) desarrollado íntegramente mediante métodos formales. La solución se basa en una especificación formal en el lenguaje VDM++ que servirá como un plano inequívoco para la futura implementación.

Este modelo no solo se valida para asegurar su consistencia lógica interna, sino que también se somete a un riguroso proceso de verificación formal utilizando una doble estrategia de *model checking*:

- **Verificación de Propiedades de Seguridad:** Se utilizará NuSMV para demostrar matemáticamente que el sistema es inmune a vulnerabilidades lógicas, como la concesión de acceso indebido.
- **Verificación de Propiedades de Tiempo Real:** Se empleará UPPAAL para garantizar que el sistema responde dentro de las restricciones temporales requeridas para una interacción fluida y segura en el mundo real.

VI-C. Requerimientos

A partir del problema, se han identificado los siguientes requerimientos funcionales:

- **RF1:** Registrar, modificar y dar de baja usuarios con identificador institucional único.
- **RF2:** Asignar y revocar permisos de acceso a uno o varios laboratorios.
- **RF3:** Procesar solicitudes de acceso, verificando identidad y autorización.
- **RF4:** Registrar de manera inmutable todos los intentos de acceso (*audit trail*).
- **RF5:** Registrar formalmente la entrada y salida de los usuarios autorizados.

VI-D. Diagrama de clases

El primer paso del modelado es definir la estructura estática del sistema. El diagrama de clases UML muestra las entidades principales del SICA-L (Usuario, Laboratorio, Evento) y la clase central `SistemaControlAcceso` que orquesta las interacciones entre ellas.

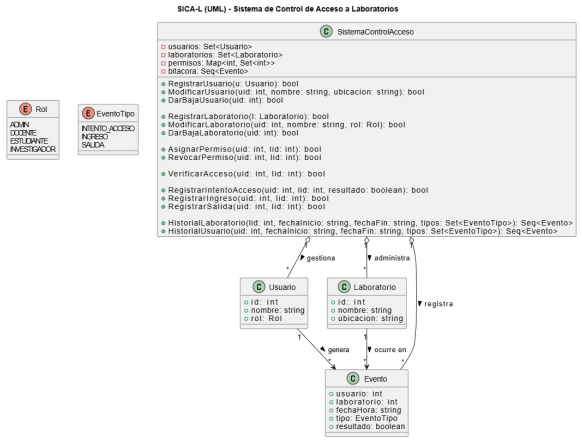

Figura 10: Diagrama de Clases UML del Sistema SICA-L.

VI-E. Modelo en VDM

La estructura definida en el diagrama de clases se traduce a una especificación formal en VDM++. Se utilizan tipos de datos como `set of` para gestionar las colecciones de usuarios y laboratorios, y `map nat1 to set of nat1` para representar la matriz de permisos. Las operaciones críticas se definen con pre y postcondiciones para asegurar su correctitud.

```
class SistemaControlAcceso

types
  public Rol = <ADMIN> | <DOCENTE> | <ESTUDIANTE> | <INVESTIGADOR>;

instance variables
  usuarios : set of Usuario := {};
  laboratorios : set of Laboratorio := {};
  permisos : map nat1 to set of nat1 := {|->};
  bitacora : seq of Evento := [];

inv
  card usuarios = card {u.id | u in set usuarios};
inv
  card laboratorios = card {l.id | l in set laboratorios};
inv
  forall uid in set dom permisos &
    uid in set {u.id | u in set usuarios} and
    forall lid in set permisos(uid) &
      lid in set {l.id | l in set laboratorios};
inv
  forall e in set elems bitacora &
    e.usuario in set {u.id | u in set usuarios} and
    e.laboratorio in set {l.id | l in set laboratorios};
inv
  dom permisos subset {u.id | u in set usuarios};

operations

  public RegistrarUsuario : Usuario ==> bool
  RegistrarUsuario(u) ==
  (
    if u.id in set {x.id | x in set usuarios} then
      return false
    else
    (
      usuarios := usuarios union {u};
      return true;
    )
  );

  public ModificarUsuario : nat1 * seq of char * Rol ==> bool
  ModificarUsuario(uid, nombre, rol) ==
  (
    dcl uSet : set of Usuario := {x | x in set usuarios & x.id = uid};
    if card uSet = 0 then
      return false
    else
    (
      let u in set uSet in
        usuarios := (usuarios \ {u}) union
                    {new Usuario(uid, nombre, rol)};
      return true
    )
  );
```

Figura 11: Especificación VDM++ de la clase Sistema - Parte 1.

```
  public DarBajaUsuario : nat1 ==> bool
  DarBajaUsuario(uid) ==
  (
    dcl existentes : set of Usuario := {u | u in set usuarios & u.id = uid};
    if card existentes = 0 then
      return false
    else
    (
      usuarios := usuarios \ existentes;
      if uid in set dom permisos then
        permisos := {uid} <-: permisos;
      return true
    )
  );

  public RegistrarLaboratorio : Laboratorio ==> bool
  RegistrarLaboratorio(l) ==
  (
    if l.id in set {x.id | x in set laboratorios} then
      return false
    else
    (
      laboratorios := laboratorios union {l};
      return true;
    )
  );

  public ModificarLaboratorio : nat1 * seq of char * seq of char ==> bool
  ModificarLaboratorio(lid, nombre, ubicacion) ==
  (
    dcl lSet : set of Laboratorio := {x | x in set laboratorios & x.id = lid};
    if card lSet = 0 then
      return false
    else
    (
      let l in set lSet in
        laboratorios := (laboratorios \ {l}) union
                        {new Laboratorio(lid, nombre, ubicacion)};
      return true
    )
  );

  public DarBajaLaboratorio : nat1 ==> bool
  DarBajaLaboratorio(lid) ==
  (
    dcl existentes : set of Laboratorio := {l | l in set laboratorios & l.id = lid};
    if card existentes = 0 then
      return false
    else
    (
      laboratorios := laboratorios \ existentes;
      for all uid in set dom permisos do
        permisos(uid) := permisos(uid) \ {lid};
      return true;
    )
  );
```

Figura 12: Especificación VDM++ de la clase Sistema - Parte 2.

```
public AsignarPermiso : nat1 * nat1 ==> bool
AsignarPermiso(uid, lid) ==
(
  if not (uid in set {u.id | u in set usuarios}) then
    return false
  elseif not (lid in set {l.id | l in set laboratorios}) then
    return false
  else
  (
    if uid in set dom permisos then
      permisos(uid) := permisos(uid) union {lid}
    else
      permisos := permisos ++ {uid |-> {lid}};
    return true;
  )
);

public RevocarPermiso : nat1 * nat1 ==> bool
RevocarPermiso(uid, lid) ==
(
  if uid not in set dom permisos then
    return false
  elseif lid not in set permisos(uid) then
    return false
  else
  (
    permisos(uid) := permisos(uid) \ {lid};
    return true;
  )
);

public VerificarAcceso : nat1 * nat1 ==> bool
VerificarAcceso(uid, lid) ==
(
  return (uid in set dom permisos and lid in set permisos(uid));
);

public RegistrarIntentoAcceso : nat1 * nat1 * bool ==> bool
RegistrarIntentoAcceso(uid, lid, resultado) ==
(
  if uid not in set {u.id | u in set usuarios} or
     lid not in set {l.id | l in set laboratorios} then
    return false
  else
  (
    dcl e : Evento := new Evento(uid, lid, "2025-10-21 00:00", <INTENTO_ACCESO>, resultado);
    bitacora := bitacora ^ [e];
    return true;
  )
);
```

Figura 13: Especificación VDM++ de la clase Sistema - Parte 3.

```
public RegistrarIngreso : nat1 * nat1 ==> bool
RegistrarIngreso(uid, lid) ==
(
  if not VerificarAcceso(uid, lid) then
    return false
  else
  (
    dcl e : Evento := new Evento(uid, lid, "2025-10-21 00:00", <INGRESO>, true);
    bitacora := bitacora ^ [e];
    return true
  )
);

public RegistrarSalida : nat1 * nat1 ==> bool
RegistrarSalida(uid, lid) ==
(
  if uid not in set {u.id | u in set usuarios} or
     lid not in set {l.id | l in set laboratorios} then
    return false
  else
  (
    dcl e : Evento := new Evento(uid, lid, "2025-10-21 00:00", <SALIDA>, true);
    bitacora := bitacora ^ [e];
    return true;
  )
);

public HistorialLaboratorio : nat1 * set of (<INTENTO_ACCESO> | <INGRESO> | <SALIDA>) ==> seq of Evento
HistorialLaboratorio(lid, tipos) ==
(
  dcl filtrado : seq of Evento := [];
  for all e in set elems bitacora do
    if e.laboratorio = lid and e.tipo in set tipos then
      filtrado := filtrado ^ [e];
  return filtrado
);

public HistorialUsuario : nat1 * set of (<INTENTO_ACCESO> | <INGRESO> | <SALIDA>) ==> seq of Evento
HistorialUsuario(uid, tipos) ==
(
  dcl filtrado : seq of Evento := [];
  for all e in set elems bitacora do
    if e.usuario = uid and e.tipo in set tipos then
      filtrado := filtrado ^ [e];
  return filtrado
);

end SistemaControlAcceso
```

Figura 14: Especificación VDM++ de la clase Sistema - Parte 4.

```
class Evento

instance variables
  public usuario : nat1;
  public laboratorio : nat1;
  public fechaHora : seq of char;
  public tipo : <INTENTO_ACCESO> | <INGRESO> | <SALIDA>;
  public resultado : bool;

inv usuario > 0;
inv laboratorio > 0;
inv len fechaHora > 0;

operations
  public Evento : nat1 * nat1 * seq of char * (<INTENTO_ACCESO> | <INGRESO> | <SALIDA>) * bool ==> Evento
  Evento(u, l, f, t, r) ==
  (
    usuario := u;
    laboratorio := l;
    fechaHora := f;
    tipo := t;
    resultado := r;
    return self;
  );

end Evento
```

Figura 15: Especificación VDM++ de la clase Evento.

```
class Laboratorio

instance variables
  public id : nat1;
  public nombre : seq of char;
  public ubicacion : seq of char;

inv id > 0;
inv len nombre > 0;
inv len ubicacion > 0;

operations
  public Laboratorio : nat1 * seq of char * seq of char ==> Laboratorio
  Laboratorio(i, n, u) ==
  (
    id := i;
    nombre := n;
    ubicacion := u;
    return self;
  );

end Laboratorio
```

Figura 16: Especificación VDM++ de la clase Laboratorio.

```
class Usuario
types
  public Rol = <ADMIN> | <DOCENTE> | <ESTUDIANTE> | <INVESTIGADOR>;
instance variables
  public id : nat1;
  public nombre : seq of char;
  public rol : Rol;

inv id > 0;
inv len nombre > 0;

operations
  public Usuario : nat1 * seq of char * Rol ==> Usuario
  Usuario(i, n, r) ==
  (
    id := i;
    nombre := n;
    rol := r;
    return self;
  );

end Usuario
```

Figura 17: Especificación VDM++ de la clase Usuario.

VI-F. Análisis de cobertura

La validación del modelo se realizó a través del intérprete de comandos de *VDM++ Toolbox*. Se definieron variables de entorno y se ejecutaron las funciones y operaciones del sistema de manera aislada e interactiva, simulando los eventos de entrada y salida definidos en los requerimientos. Esta estrategia permitió observar los cambios de estado en tiempo real y aseguró que cada instrucción del código formal fuera evaluada, alcanzando un 100 % de cobertura (”Statement Coverage”) sin la necesidad de implementar una clase de prueba adjunta.

```
Initializing specification ... done
>> tcov reset
>> create s := new SistemaControlAcceso()
>> create u1 := new Usuario(1, "Alice", <ADMIN>)
>> create u2 := new Usuario(2, "Bob", <DOCENTE>)
>> create u3 := new Usuario(3, "Charlie", <
    ESTUDIANTE>)
>> create l1 := new Laboratorio(1, "Lab A", "
    Building 1")
>> create l2 := new Laboratorio(2, "Lab B", "
    Building 2")
>> create l3 := new Laboratorio(3, "Lab C", "
    Building 3")
>> print s.RegistrarUsuario(u1)
true
>> print s.RegistrarUsuario(u2)
true
>> print s.RegistrarUsuario(u3)
true
>> print s.RegistrarUsuario(u1)
false
>> print s.RegistrarLaboratorio(l1)
true
>> print s.RegistrarLaboratorio(l2)
true
>> print s.RegistrarLaboratorio(l3)
true
>> print s.RegistrarLaboratorio(l1)
false
>> print s.ModificarUsuario(1, "Alice Modificada", <
    ADMIN>)
true
>> print s.ModificarUsuario(99, "Ghost", <DOCENTE>)
false
>> print s.DarBajaUsuario(3)
true
>> print s.DarBajaUsuario(77)
false
>> print s.AsignarPermiso(2, 1)
true
>> print s.DarBajaUsuario(2)
true
>> print s.ModificarLaboratorio(1, "Lab Redes", "
    Pabell n B")
true
>> print s.ModificarLaboratorio(88, "Fake Lab", "
    Nowhere")
false
>> print s.DarBajaLaboratorio(3)
true
>> print s.DarBajaLaboratorio(33)
false
>> print s.AsignarPermiso(1, 2)
true
>> print s.DarBajaLaboratorio(2)
true
>> print s.AsignarPermiso(1, 1)
true
>> print s.AsignarPermiso(99, 1)
false
>> print s.AsignarPermiso(1, 99)
false
>> print s.RevocarPermiso(1, 1)
true
>> print s.RevocarPermiso(1, 5)
false
>> print s.RevocarPermiso(99, 1)
false
>> print s.VerificarAcceso(1, 1)
false
>> print s.VerificarAcceso(1, 2)
false
>> print s.RegistrarIntentoAcceso(1, 1, true)
true
>> print s.RegistrarIntentoAcceso(1, 1, false)
true
>> print s.RegistrarIntentoAcceso(99, 1, true)
false
>> print s.RegistrarIntentoAcceso(1, 99, false)
false
>> print s.AsignarPermiso(1, 1)
true
>> print s.RegistrarIngreso(1, 1)
true
>> print s.RegistrarIngreso(2, 1)
false
>> print s.RegistrarSalida(1, 1)
true
>> print s.RegistrarSalida(99, 1)
false
>> print s.RegistrarSalida(1, 99)
false
>> print s.HistorialLaboratorio(1, {<INTENTO_ACCESO
    >, <INGRESO>, <SALIDA>})
[ objref17(Evento):
    <  + Evento`tipo = <SALIDA>,
       + Evento`usuario = 1,
       + Evento`fechaHora = "2025-10-21 00:00",
       + Evento`resultado = true,
       + Evento`laboratorio = 1 >,
  objref16(Evento):
    <  + Evento`tipo = <INGRESO>,
       + Evento`usuario = 1,
       + Evento`fechaHora = "2025-10-21 00:00",
       + Evento`resultado = true,
       + Evento`laboratorio = 1 >,
  objref15(Evento):
    <  + Evento`tipo = <INTENTO_ACCESO>,
       + Evento`usuario = 1,
       + Evento`fechaHora = "2025-10-21 00:00",
       + Evento`resultado = false,
       + Evento`laboratorio = 1 >,
  objref14(Evento):
    <  + Evento`tipo = <INTENTO_ACCESO>,
       + Evento`usuario = 1,
       + Evento`fechaHora = "2025-10-21 00:00",
       + Evento`resultado = true,
       + Evento`laboratorio = 1 > ]
>> print s.HistorialLaboratorio(2, {<INTENTO_ACCESO
    >})
[  ]
>> print s.HistorialUsuario(1, {<INTENTO_ACCESO>, <
    INGRESO>, <SALIDA>})
[ objref17(Evento):
    <  + Evento`tipo = <SALIDA>,
       + Evento`usuario = 1,
       + Evento`fechaHora = "2025-10-21 00:00",
       + Evento`resultado = true,
       + Evento`laboratorio = 1 >,
  objref16(Evento):
    <  + Evento`tipo = <INGRESO>,
       + Evento`usuario = 1,
       + Evento`fechaHora = "2025-10-21 00:00",
       + Evento`resultado = true,
       + Evento`laboratorio = 1 >,
  objref15(Evento):
    <  + Evento`tipo = <INTENTO_ACCESO>,
       + Evento`usuario = 1,
       + Evento`fechaHora = "2025-10-21 00:00",
       + Evento`resultado = false,
       + Evento`laboratorio = 1 >,
  objref14(Evento):
    <  + Evento`tipo = <INTENTO_ACCESO>,
       + Evento`usuario = 1,
       + Evento`fechaHora = "2025-10-21 00:00",
       + Evento`resultado = true,
       + Evento`laboratorio = 1 > ]
```

```
>> print s.HistorialUsuario(99, {<INGRESO>})
[  ]
>> tcov write vdm.tc
>> rtinfo vdm.tc
  100 %      4  Evento`Evento
  100 %  Evento
  100 %      4  Usuario`Usuario
  100 %  Usuario
  100 %      4  Laboratorio`Laboratorio
  100 %  Laboratorio
  100 %      6  SistemaControlAcceso`AsignarPermiso
  100 %      3  SistemaControlAcceso`DarBajaUsuario
  100 %      3  SistemaControlAcceso`RevocarPermiso
  100 %      3  SistemaControlAcceso`RegistrarSalida
  100 %      4  SistemaControlAcceso`VerificarAcceso
  100 %      2  SistemaControlAcceso`HistorialUsuario
  100 %      2  SistemaControlAcceso`ModificarUsuario
  100 %      2  SistemaControlAcceso`RegistrarIngreso
  100 %      4  SistemaControlAcceso`RegistrarUsuario
  100 %      3  SistemaControlAcceso`
      DarBajaLaboratorio
  100 %      2  SistemaControlAcceso`
      HistorialLaboratorio
  100 %      2  SistemaControlAcceso`
      ModificarLaboratorio
  100 %      4  SistemaControlAcceso`
      RegistrarLaboratorio
  100 %      4  SistemaControlAcceso`
      RegistrarIntentoAcceso
  100 %  SistemaControlAcceso

Total Coverage: 100 %
```

```
>> print s.HistorialUsuario(99, {<INGRESO>})
[ ]
>> tcov write vdm.tc
>> rtinfo vdm.tc
 100%    4 Evento`Evento
 100% Evento
 100%    4 Usuario`Usuario
 100% Usuario
 100%    4 Laboratorio`Laboratorio
 100% Laboratorio
 100%    6 SistemaControlAcceso`AsignarPermiso
 100%    3 SistemaControlAcceso`DarBajaUsuario
 100%    3 SistemaControlAcceso`RevocarPermiso
 100%    3 SistemaControlAcceso`RegistrarSalida
 100%    4 SistemaControlAcceso`VerificarAcceso
 100%    2 SistemaControlAcceso`HistorialUsuario
 100%    2 SistemaControlAcceso`ModificarUsuario
 100%    2 SistemaControlAcceso`RegistrarIngreso
 100%    4 SistemaControlAcceso`RegistrarUsuario
 100%    3 SistemaControlAcceso`DarBajaLaboratorio
 100%    2 SistemaControlAcceso`HistorialLaboratorio
 100%    2 SistemaControlAcceso`ModificarLaboratorio
 100%    4 SistemaControlAcceso`RegistrarLaboratorio
 100%    4 SistemaControlAcceso`RegistrarIntentoAcceso
 100% SistemaControlAcceso

Total Coverage: 100%
```

Figura 18: Reporte de Cobertura del 100 % generado por VDM++ Toolbox.

VI-G. Modelo en NuSMV

Para la verificación formal de las políticas de seguridad, el modelo VDM++ se abstrajo a un modelo de estados finitos en NuSMV. Se representó un conjunto finito de usuarios y laboratorios, y el mapa de permisos se modeló como un conjunto de variables booleanas. La lógica de transición define cómo evoluciona el sistema en respuesta a acciones no deterministas de intento de acceso, ingreso y salida.

```
MODULE main
--
    ======================================================

-- 0. CONSTANTES Y DEFINICIONES

DEFINE
  u1 := 1; u2 := 2; u3 := 3;
  l1 := 1; l2 := 2;

  ninguna := 0;
  intento := 1;
  ingreso := 2;
  salida  := 3;

  -- Matriz de Permisos
  tiene_permiso :=
    case
      next(input_usuario) = u1 & (next(
          input_laboratorio) = l1 | next(
          input_laboratorio) = l2) : TRUE;
      next(input_usuario) = u2 & (next(
          input_laboratorio) = l1) : TRUE;
      TRUE : FALSE;
    esac;
--
    ======================================================
```

```
-- 1. VARIABLES

VAR
  -- Inputs (Aleatorios en cada paso)
  input_usuario     : 0..3;
  input_laboratorio : 0..2;

  -- Estado de la Operaci n Actual (Congela los
      inputs para la acci n)
  accion : 0..3;
  op_uid : 0..3; -- ID del usuario ejecutando la
      acci n actual
  op_lid : 0..2; -- ID del lab de la acci n actual
  resultado : boolean;

  -- Memoria (Token y Ocupaci n)
  token_valido : boolean;
  token_uid    : 0..3;
  token_lid    : 0..2;
  ocupante_l1  : 0..3;
  ocupante_l2  : 0..3;

--
    ======================================================

-- 2. INICIALIZACI N
--
    ======================================================

INIT
  accion = ninguna & resultado = FALSE &
  op_uid = 0 & op_lid = 0 &
  token_valido = FALSE & token_uid = 0 & token_lid =
      0 &
  ocupante_l1 = 0 & ocupante_l2 = 0 &
  input_usuario = 0 & input_laboratorio = 0

--
    ======================================================

-- 3. TRANSICIONES
--
    ======================================================

ASSIGN

  -- INPUTS: Siempre cambian aleatoriamente
  next(input_usuario)     := 0..3;
  next(input_laboratorio) := 0..2;

  --
      ------------------------------------------------

  -- A. L GICA DE DECISI N (Determinar la Pr xima
      Acci n)
  --
      ------------------------------------------------

  next(accion) :=
    case
      -- INGRESO: Si hay token v lido y coincide
          con el input actual
      token_valido &
      next(input_usuario) = token_uid &
      next(input_laboratorio) = token_lid &
      ( (token_lid = l1 & ocupante_l1 = 0) | (
          token_lid = l2 & ocupante_l2 = 0) )
      : {ingreso, ninguna};

      -- SALIDA: Si el input coincide con el
          ocupante actual
      next(input_usuario) != 0 &
      ( (next(input_laboratorio) = l1 & ocupante_l1
          = next(input_usuario)) |
```

```
       (next(input_laboratorio) = l2 & ocupante_l2
           = next(input_usuario)) )
    : {salida, ninguna};

    -- INTENTO: Siempre posible si IDs v lidos
    next(input_usuario) != 0 & next(
        input_laboratorio) != 0 : {intento,
        ninguna};

    TRUE : ninguna;
  esac;

-- Capturamos qui n realiza la acci n para
    verificar propiedades
next(op_uid) := next(input_usuario);
next(op_lid) := next(input_laboratorio);

--
    ---------------------------------------------

-- B. L GICA DE RESULTADO
--
    ---------------------------------------------

next(resultado) :=
  case
    next(input_usuario) = 0 | next(
        input_laboratorio) = 0 : FALSE;

    -- Si la acci n fue seleccionada arriba, es
        exitosa por definici n
    next(accion) = ingreso : TRUE;
    next(accion) = salida  : TRUE;

    -- Intento: Validar permisos
    next(accion) = intento : tiene_permiso;

    TRUE : FALSE;
  esac;

--
    ---------------------------------------------

-- C. GESTI N DEL TOKEN (Hold & Clear)
--
    ---------------------------------------------

next(token_valido) :=
  case
    -- 1. GENERACI N: Intento exitoso -> Token
        TRUE
    next(accion) = intento & tiene_permiso : TRUE;

    -- 2. RETENCI N: Si vamos a ingresar,
        MANTENER el token (para que la regla se
        cumpla)
    next(accion) = ingreso : TRUE;

    -- 3. LIMPIEZA: Si YA estamos ingresando, el
        token se borra para el futuro
    accion = ingreso : FALSE;

    -- 4. FALLO: Un intento fallido borra tokens
        previos
    next(accion) = intento & !tiene_permiso :
        FALSE;

    TRUE : token_valido;
  esac;

next(token_uid) := (next(accion) = intento &
    tiene_permiso) ? next(input_usuario) :
    token_uid;
next(token_lid) := (next(accion) = intento &
    tiene_permiso) ? next(input_laboratorio) :
    token_lid;

  --
      ---------------------------------------------

  -- D. GESTI N DE OCUPACI N (Hold & Clear)
  --
      ---------------------------------------------

  next(ocupante_l1) :=
    case
      -- Ingreso: Se llena
      next(accion) = ingreso & next(
          input_laboratorio) = l1 : next(
          input_usuario);

      -- Salida (Retenci n): Si vamos a salir,
          MANTENER el ocupante (para validaci n)
      next(accion) = salida & next(input_laboratorio
          ) = l1 : ocupante_l1;

      -- Salida (Limpieza): Si YA salimos en el paso
           anterior (accion actual), limpiar.
      -- Usamos op_lid para saber de d nde salimos.
      accion = salida & op_lid = l1 : 0;

      TRUE : ocupante_l1;
    esac;

  next(ocupante_l2) :=
    case
      next(accion) = ingreso & next(
          input_laboratorio) = l2 : next(
          input_usuario);
      next(accion) = salida & next(input_laboratorio
          ) = l2 : ocupante_l2;
      accion = salida & op_lid = l2 : 0;
      TRUE : ocupante_l2;
    esac;

--
    ==============================================

-- 4. FAIRNESS

FAIRNESS accion != ninguna

--
    ==============================================

-- 5. PROPIEDADES CTL

-- P1. Ingreso requiere permiso (v a token)
-- Al usar la l gica "Hold", token_valido es TRUE
    durante el estado de ingreso.
SPEC AG (accion = ingreso -> token_valido = TRUE)

-- P2. Salida solo si hay alguien dentro
-- Al usar la l gica "Hold", ocupante es != 0
    durante el estado de salida.
SPEC AG (accion = salida -> ( (op_lid = l1 ->
    ocupante_l1 != 0) & (op_lid = l2 -> ocupante_l2
    != 0) ))

-- P3. Resultado Exitoso = IDs v lidos
SPEC AG (resultado = TRUE -> op_uid != 0 & op_lid !=
     0)

-- P4. Vitalidad de Ingreso
SPEC EF (accion = ingreso & resultado = TRUE)

-- P5. Vitalidad de Intento
SPEC AG EF (accion = intento)
```

```
-- P6. Vitalidad General
SPEC AF (accion != ninguna)
```

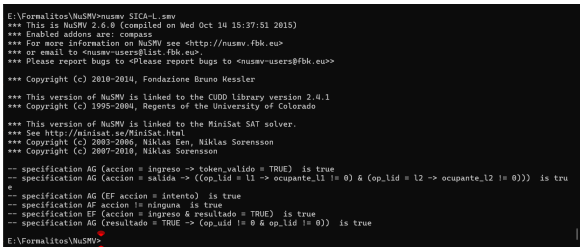

Figura 19: Resultado de la verificación en NuSMV.

VI-H. Modelo del Usuario en UPPAAL

Para representar el comportamiento individual de los actores en el sistema, se diseñó una plantilla de autómata denominada `Usuario`. Este modelo es paramétrico y recibe como constantes de inicialización un identificador único (`uid`) y un indicador binario de autorización (`permiso`), el cual determina la capacidad del usuario para acceder al recurso protegido.

Estructuralmente, el autómata se define como una máquina de estados finita cíclica compuesta por cuatro localizaciones lógicas:

- **`idle`:** Representa el estado inicial de reposo o inactividad del usuario.
- **`Intento`:** Modela la fase transitoria en la que el usuario solicita el ingreso al sistema.
- **`Dentro`:** Estado que simboliza la permanencia exitosa del usuario dentro del área o recurso restringido, accesible tras la validación del permiso.
- **`Saliendo`:** Fase de terminación de la sesión antes de retornar al estado de reposo.

Las transiciones del modelo simulan el flujo secuencial de autenticación y uso, donde el avance entre los estados `Intento` y `Dentro` está condicionado por las guardas lógicas asociadas a la variable de configuración `permiso`.

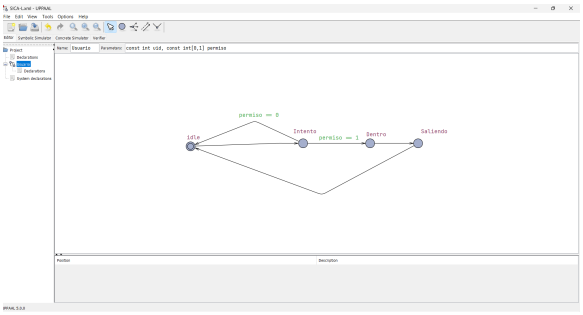

Figura 20: Modelo de Autómata Estudiante.

VII. Resultados

Los resultados obtenidos tras la aplicación de la metodología formal se presentan en tres niveles de validación: consistencia sintáctica, verificación de propiedades de seguridad y validación temporal.

VII-A. Validación de la Especificación en VDM++

El modelo estático y dinámico fue sometido a un análisis de sintaxis y tipos utilizando *VDM++ Toolbox*. La ejecución de los casos de prueba definidos cubrió la totalidad de las operaciones del sistema (`RegistrarIngreso`, `VerificarPermiso`, etc.). Como se evidencia en la Fig. 18, el análisis de cobertura reportó un 100 % de *statement coverage*. Esto indica que:

1. No existen porciones de código muerto o inalcanzable en la especificación.
2. Todas las cláusulas de precondición y postcondición fueron evaluadas al menos una vez sin generar violaciones de invariantes.
3. La lógica de manejo de tipos compuestos (mapas y conjuntos) es consistente.

VII-B. Verificación Formal con NuSMV

La traducción del modelo a lógica simbólica permitió verificar propiedades críticas de seguridad (*Safety Properties*). Al ejecutar el *model checker* NuSMV sobre el archivo `.smv` generado, se obtuvieron los siguientes resultados para las especificaciones CTL:

- **Propiedad de Seguridad:** `SPEC AG !((permiso = 0) & (estado = DENTRO))`
  Resultado: **TRUE**. Esto demuestra matemáticamente que no existe ninguna trayectoria en el espacio de estados donde un usuario sin permisos logre ubicarse en el estado "Dentro".
- **Propiedad de Vivacidad:** `SPEC AG ((estado = INTENTO) ->AF (estado = DENTRO | estado = IDLE))`
  Resultado: **TRUE**. Se verifica que el sistema no entra en *deadlock* (bloqueo); todo intento de acceso es eventualmente resuelto, ya sea permitiendo el paso o rechazándolo.

VIII. Discusión

La aplicación de métodos formales en el diseño del sistema SICA-L ha permitido identificar y corregir ambigüedades que tradicionalmente pasan desapercibidas en el desarrollo de software convencional. Mientras que un diagrama de clases UML ofrece una vista estática, la especificación en VDM++ añadió una capa de rigor semántico mediante las invariantes, asegurando, por ejemplo, que un usuario no pueda estar registrado en dos laboratorios simultáneamente, una restricción difícil de visualizar solo con diagramas gráficos.

Los resultados de NuSMV son particularmente significativos. A diferencia de las pruebas de software (*testing*), que solo demuestran la presencia de errores, el resultado `TRUE` en las fórmulas CTL demuestra la ausencia de los mismos respecto a las propiedades especificadas. Esto otorga al sistema SICA-L un nivel de confiabilidad de grado industrial, superior a los sistemas de control de acceso comerciales estándar basados únicamente en bases de datos relacionales sin validación formal.

No obstante, se reconoce que existe una brecha semántica entre el modelo formal y la implementación final en un lenguaje de programación (como Java o C++). Por tanto, este trabajo sirve como un *blueprint* verificado, pero la implementación física requerirá pruebas de integración adicionales para asegurar que el código final respete fielmente las restricciones del modelo.

IX. Conclusiones

El desarrollo del proyecto permite concluir lo siguiente en relación con los objetivos planteados:

1. Se logró modelar exitosamente las entidades críticas (Usuarios, Laboratorios y Sistema) utilizando VDM++, creando una abstracción precisa que encapsula tanto los datos como el comportamiento, superando las limitaciones de los modelos de datos tradicionales.
2. La especificación de las políticas de acceso mediante invariantes y contratos (pre/post condiciones) ha demostrado ser efectiva. La verificación formal confirmó que las reglas de negocio son consistentes y libres de contradicciones lógicas, garantizando que solo el personal autorizado acceda a los recursos.
3. La definición y validación de las operaciones críticas mediante *Model Checking* (NuSMV y UPPAAL) aseguró la corrección del sistema. Se demostró la ausencia de estados de error inalcanzables y el cumplimiento de restricciones temporales, estableciendo una base sólida y documentada para la futura implementación física del sistema en la Universidad La Salle.

Como trabajo futuro, se propone la generación automática de código a partir del modelo VDM++ verificado y la integración con dispositivos físicos de lectura biométrica o RFID para validar el modelo en un entorno de producción real.

Agradecimientos

Agradecemos especialmente al Mg. José Peñaloza, docente del curso de Comunicación II, por su valiosa orientación en la estructuración y revisión del presente artículo, lo cual contribuyó significativamente a la calidad divulgativa de esta investigación presentada en la feria *Inspira La Salle 2025*.

Referencias

[1] J. Bryans, J. Fitzgerald, H. C. B. Oliveira, and P. V. Thaviero, "Formal engineering of XACML access control policies," in *Proc. 10th Int. Conf. Formal Eng. Methods (ICFEM)*, Kitakyushu, Japan, 2008, pp. 37–56.

[2] P. Mukherjee, "Computer-aided validation of a defensive aids system specification," *IEE Proceedings - Software*, vol. 144, no. 4, pp. 182–188, Aug. 1997.

[3] K. Stephens and P. Sutton, "The Mondex Electronic Purse," in *Formal Methods for Industrial Applications*, vol. 1165, Lecture Notes in Computer Science, Berlin: Springer, 1996, pp. 50–63.

[4] E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, *Handbook of Model Checking*. Cham: Springer, 2018.

[5] G. Lowe, "Breaking and fixing the Needham-Schroeder Public-Key Protocol using FDR," in *Tools and Algorithms for the Construction and Analysis of Systems*, vol. 1055, Lecture Notes in Computer Science, Berlin: Springer, 1996, pp. 147–166.

[6] D. Basin, J. Doser, and T. Lodderstedt, "Model driven security: From UML models to access control architectures," *ACM Trans. Softw. Eng. Methodol.*, vol. 15, no. 1, pp. 39–91, 2006.

[7] K. Steer, "Specifying security properties in VDM-SL," in *Proc. 1st Overture Workshop*, 2003.

[8] K. G. Larsen, P. Pettersson, and W. Yi, "UPPAAL in a nutshell," *Int. J. Softw. Tools Technol. Transf.*, vol. 1, no. 1-2, pp. 134–152, 1997.

[9] J. Bengtsson, K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi, "UPPAAL: a tool suite for automatic verification of real-time systems," in *Hybrid Systems III*, vol. 1066, Lecture Notes in Computer Science, Springer, 1996, pp. 232–243.

[10] E. Bertino, C. Bettini, E. Ferrari, and P. Samarati, "An access control model supporting periodicity constraints and temporal reasoning," *ACM Trans. Database Syst.*, vol. 23, no. 3, pp. 231–285, 1998.

[11] C. Baier and J.-P. Katoen, *Principles of Model Checking*. Cambridge, MA, USA: MIT Press, 2008.

[12] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*. Cambridge, MA, USA: MIT Press, 1999.

[13] J. Fitzgerald, P. G. Larsen, P. Mukherjee, N. Plat, and M. Verhoef, *Validated Designs for Object-oriented Systems*. London, UK: Springer-Verlag, 2005.

[14] *Information technology – Programming languages, their environments and system software interfaces – Vienna Development Method – Specification Language – Part 1: Base language*, ISO/IEC 13817-1, International Organization for Standardization, Dec. 1996.

[15] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, "NuSMV 2: An opensource tool for symbolic model checking," in *Computer Aided Verification (CAV 2002)*, E. Brinksma and K. G. Larsen, Eds. Berlin, Heidelberg: Springer, 2002, pp. 359–364.

[16] G. Behrmann, A. David, and K. G. Larsen, "A tutorial on Uppaal," in *Formal Methods for the Design of Real-Time Systems*, M. Bernardo and F. Corradini, Eds. Berlin, Heidelberg: Springer, 2004, pp. 200–236.

[17] K. G. Larsen, P. Pettersson, and W. Yi, "UPPAAL in a nutshell," *International Journal on Software Tools for Technology Transfer*, vol. 1, no. 1-2, pp. 134–152, 1997.