ErisML: A Formal Modeling Language for Governed Foundation-Model Agents in Pervasive Computing Environments

Andrew Bond, Senior Member, IEEE
Department of Computer Engineering, San José State University

andrew.bond@sjsu.edu

Table of Contents

Abstract

Foundation models and agentic systems are rapidly moving into pervasive computing environments such as homes, hospitals, factories, and campuses. These settings couple heterogeneous sensors with resource-constrained edge devices and safety-critical actuators, and they expose agents directly to human norms, institutions, and regulations. While the machine-learning community has refined architectures for multimodal reasoning, tool-augmented planning, and long-horizon control, the specification layer remains fragmented: planners, reinforcement-learning frameworks, agent-based simulators, and deontic logics each describe part of the problem but offer no shared substrate for stating what the world is, who acts in it, what they want, what is allowed, and how decisions evolve over time.

This paper introduces *ErisML*, a formal modeling language for foundation-model-enabled agents in pervasive environments. ErisML provides a single, machine-interpretable and human-legible representation of (i) environment state and dynamics, (ii) agents and their capabilities and beliefs, (iii) intents and utilities, (iv) norms (permissions, obligations, prohibitions, sanctions), and (v) multi-agent strategic interaction. We define a concrete syntax, a formal grammar, denotational semantics, and an execution model that treats norms as first-class constraints on action, introduces longitudinal safety metrics such as Norm Violation Rate (NVR) and Alignment Drift Velocity (ADV), and supports compilation to planners, verifiers, and simulators.

We describe a reference toolchain (parser, type checker, compiler, verifier, and monitor), show how ErisML integrates with edge-deployed foundation models and federated learning, and present a worked example together with application sketches in healthcare, smart-campus mobility, and industrial maintenance. We argue that ErisML can serve as a common substrate for technical integration and regulatory governance, enabling legible, auditable, and governable ambient systems.

Pervasive computing, foundation models, AI agents, modeling languages, multi-agent systems, normative systems, safety, alignment, human–AI interaction, federated learning, edge optimization.

Introduction

Pervasive or “ambient” intelligence—computation woven into everyday environments—is no longer speculative. Home assistants orchestrate appliances, clinical monitors run continuously in hospital wards, industrial plants use predictive agents to schedule maintenance, and smart campuses coordinate shuttles, delivery robots, and building systems. Increasingly, these systems are built around foundation models (FMs) that can interpret multimodal inputs, reason with tools, and drive multi-step policies.

However, the *specification substrate* lags behind the modeling substrate. Practitioners routinely combine textual prompts and instructions, handcrafted rules, planner domain files (e.g., PDDL), reinforcement-learning reward functions, and ad-hoc policy gates. This patchwork makes it difficult to answer basic questions such as:

- What is the environment, as the system “sees” it?
- Which agents exist, and what are their capabilities and beliefs?

- What are the agents optimizing, and how are trade-offs resolved?
- Which norms are in force, and how do they constrain behavior?
- How do we measure drift, violations, and safety over time?

We argue that pervasive, FM-enabled systems require a unified modeling language that elevates environment, agency, intent, norms, and dynamics to first-class constructs. *ErisML* is such a language.

Contributions

Building on an earlier, less formal draft, this paper:

- Defines a core syntax and *formal grammar* for ErisML with four primary blocks: `environment`, `agent`, `norms`, and `dynamics`.
- Provides a denotational semantics mapping ErisML models to norm-constrained multi-agent decision processes, and introduces longitudinal safety metrics (NVR, ADV, and a stability–plasticity margin).
- Describes an execution model in which ErisML sits above FM inference, mediating tool calls and actuations.
- Outlines a reference toolchain including parser, type checker, compiler, verifier, and monitor.
- Includes a *worked mathematical example* of a tiny ErisML model with a fully spelled-out state space, action space, and norms.

Outline

Section 2 describes design goals and core abstractions. Section 3 gives the formal grammar. Section 4 covers the formal semantics. Section 5 presents a worked example. Section 6 describes integration with FMs and edge systems. Section 7 discusses toolchain design. Sections 8 and 9 address evaluation and applications, followed by related work and conclusions.

Design Requirements and Conceptual Overview

Design Goals

ErisML is guided by the following goals:

Provide one language that can express environment state and dynamics, agents and their capabilities and beliefs, intents and utilities, norms, and multi-agent interactions.

Compile cleanly to planners, RL environments, and verifiers, while remaining readable to engineers, auditors, and regulators.

Support cooperative, competitive, and mixed-motive scenarios with heterogeneous agents (humans, robots, services, institutions).

Offer precise semantics for state evolution, utility aggregation, and normative effects, enabling rigorous analysis and tool interoperability.

Core Abstractions

Conceptually, an ErisML model consists of:

- *Environment* E: object types, instances, state variables, and dynamics.
- *Agents* $\mathcal{A}$: each with capabilities, belief models, and links to policies and tools.
- *Intents* G: vector-valued utilities and scalarization rules.
- *Norms* N: permissions, obligations, prohibitions, and sanctions.
- *Dynamics* (T, R): joint actions, transition kernels, and reward/utility composition.

These induce a norm-constrained multi-agent decision process:

$$\mathcal{M} = \langle S, \{A_i\}_{i\in\mathcal{A}}, T, \{U_i\}_{i\in\mathcal{A}}, N\rangle,$$

where S is the state space, A_i is the action set for agent i, T is a (possibly stochastic) transition kernel, U_i are utility mappings, and N is a set of norms constraining feasible actions.

Concrete Syntax and Formal Grammar

This section specifies the concrete syntax of ErisML via an abstract grammar. The grammar is intentionally compact; a production-quality implementation may refine it further (e.g., richer types, modules).

Lexical Structure

We assume a conventional lexical layer with:

- **Identifiers** (`Identifier`): sequences of letters, digits, and underscores, not starting with a digit.
- **Numbers** (`Number`): real or integer literals.

- **Keywords:** `environment`, `agent`, `norms`, `dynamics`, `objects`, `state`, `capabilities`, `beliefs`, `intents`, `constraints`, `permission`, `prohibition`, `obligation`, `sanction`, `reward`, `joint_action`, `updates`, `if`, `unless`.
- **Symbols:** braces `{,}`, parentheses `(,)`, brackets, arrows `->`, colon `:`, semicolon `;`, comma `,`, etc.

Abstract Grammar

We present the grammar in an EBNF-like style. Nonterminals are capitalized; terminals appear in `typewriter`.

```
            Model ::=  EnvironmentBlock  AgentBlock+  NormBlock?  DynamicsBlock?
 EnvironmentBlock ::=  environment Identifier { EnvBody }
          EnvBody ::=  ObjectsDecl StateDecl EnvDynDecl?
      ObjectsDecl ::=  objects: ObjList ;
          ObjList ::=  Identifier (, Identifier)*
        StateDecl ::=  state: StateVarDecl*
     StateVarDecl ::=  Identifier : TypeExpr ;
         TypeExpr ::=  BaseType | MappingType
         BaseType ::=  bool | int | real | (TypeExpr , TypeExpr)
      MappingType ::=  Identifier -> BaseType
       EnvDynDecl ::=  dynamics: EnvRule*
          EnvRule ::=  Identifier ( ParamList? ) UpdateBlock
        ParamList ::=  ParamDecl (, ParamDecl)*
        ParamDecl ::=  Identifier : TypeExpr
      UpdateBlock ::=  updates UpdateStmt+
       UpdateStmt ::=  LValue = Expr ;
           LValue ::=  Identifier | Identifier[Expr]
       AgentBlock ::=  agent Identifier { AgentBody }
        AgentBody ::=  CapabilitiesDecl BeliefsDecl IntentsDecl ConstraintsDecl?
 CapabilitiesDecl ::=  capabilities: Identifier* ;
      BeliefsDecl ::=  beliefs: BeliefExpr* ;
      IntentsDecl ::=  intents: IntentExpr* ;
  ConstraintsDecl ::=  constraints: ConstraintExpr* ;
       BeliefExpr ::=  Identifier | Expr
       IntentExpr ::=  Identifier(ArgList?)
   ConstraintExpr ::=  Expr
        NormBlock ::=  norms Identifier { NormRule* }
         NormRule ::=  PermissionRule | ProhibitionRule | ObligationRule | SanctionRule
   PermissionRule ::=  permission: Expr ;
  ProhibitionRule ::=  prohibition: Expr ;
   ObligationRule ::=  obligation: Expr ;
     SanctionRule ::=  sanction: Expr ;
    DynamicsBlock ::=  dynamics { JointActionDecl RewardDecl }
  JointActionDecl ::=  joint_action { JointTerm+ }
        JointTerm ::=  Identifier.Identifier -> cost Number ;
       RewardDecl ::=  reward { RewardTerm+ }
       RewardTerm ::=  Identifier : UtilityExpr ;
      UtilityExpr ::=  Identifier(ArgList?) | Expr
          ArgList ::=  Expr (, Expr)*
             Expr ::=  Identifier | Number | Expr Op Expr | (Expr) | Identifier[Expr]
               Op ::=  + | - | * | / | == | != | < | > | <= | >=
```

This grammar is intentionally compact; implementations may add syntactic sugar (e.g., set literals, derived predicates) and a module system.

Formal Semantics

We sketch the denotational semantics of ErisML models. Time is discrete for simplicity.

Environment and State Space

Let $\mathcal{O}$ be the set of declared object types. For each $O \in \mathcal{O}$, let Inst(O) be a finite index set of instances. Let $\mathcal{V}$ be the set of state variables. Each variable $v \in \mathcal{V}$ has an associated type mapping τ_v; for example, `load: PowerNodes -> real` defines

$$\tau_{\text{load}}\text{:Inst(PowerNodes)} \to \mathbb{R}.$$

Definition 1 (State Space). *A concrete state is an assignment* $s = \{v \mapsto f_v \mid v \in \mathcal{V},\ f_v \in \text{Interp}(\tau_v)\}$*, where* $\text{Interp}(\tau_v)$ *is the set of functions consistent with* τ_v*. The state space* S *is the set of all such assignments.*

Environment dynamics rules define primitive state transformers $\delta_r: S \times \text{Args}_r \to S$, which are composed into a transition kernel T based on which joint actions are taken.

Agents, Actions, and Observations

Let $\mathcal{A}$ be the set of agents declared via `agent` blocks. For each $i \in \mathcal{A}$, the `capabilities` section induces an *action space* A_i by ground instantiation of action schemas with arguments. The global joint action space is:

$$A = \prod_{i \in \mathcal{A}} A_i.$$

Agents may be partially informed. Each agent i has an observation space Ω_i and a belief space $\Delta(S)$ (the set of probability distributions over S). A belief update operator Update_i maps prior beliefs, actions, and observations to posterior beliefs.

Intents and Utilities

The `intents` section provides a multi-dimensional utility vector $U_i: S \times A \to \mathbb{R}^{d_i}$ for each agent i. A scalarization operator Scalarize_i and parameters θ_i define a scalar utility:

$$\widetilde{U}_i(s, a) = \text{Scalarize}_i(U_i(s, a), \theta_i).$$

Norms and Norm-Gated Actions

Norms introduce hard and soft constraints.

Definition 2 (Norm-Gated Actions). *Let* $\phi_{\text{norm}}: S \times A \to \{\text{True}, \text{False}\}$ *be a normative feasibility predicate derived from* `prohibition` *and hard* `obligation` *rules. The norm-permissible action set at state s is* $A_N(s) = \{a \in A \mid \phi_{\text{norm}}(s, a) = \text{True}\}$.

Soft norms (e.g., advisories) and sanctions are incorporated into penalty terms $g_k(s, a)$ and folded into the effective utility via Lagrange multipliers.

Agent Objectives and Dynamics

Given a joint policy $\pi = (\pi_i)_{i\in\mathcal{A}}$, with $a_t \sim \pi(\cdot|\ h_t)$ constrained to $A_N(s_t)$, the induced process evolves as:

$$s_{t+1} \sim T(s_t, a_t),$$

and each agent i optimizes

$$J_i(\pi) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t\, \widetilde{U}_i(s_t, a_t)\right]$$

subject to $a_t \in A_N(s_t)$ and any additional constraints (e.g., bounded probability of certain norm violations).

Longitudinal Safety Metrics

Over a finite window $W = \{t_0, \dots, t_1\}$, with V norm-violation attempts (actions proposed but blocked by norms or triggering sanctions), the Norm Violation Rate is:

$$\mathrm{NVR}(W) = \frac{V}{|W|}.$$

Alignment Drift Velocity can be defined in terms of utility or policy changes, e.g., for policies:

$$\mathrm{ADV}_\pi = \frac{D(\pi_t, \pi_{t-\Delta})}{\Delta},$$

for a suitable divergence D. A stability–plasticity margin compares performance on critical tasks before and after updates.

Worked Example: Tiny ErisML Model

We now give a minimal ErisML model and spell out its state space, action space, and norms formally. This example is intentionally small but complete.

Syntax of the Tiny Model

Consider a two-room environment with a mobile robot and a human.

```
environment TinyHome {
  objects: Human, Robot, Room;

  state:
    location: (Human | Robot) -> Room;
    light_on: Room -> bool;

  dynamics:
    move_robot(from: Room, to: Room)
      updates location[Robot] = to;
    toggle_light(r: Room)
      updates light_on[r] = !light_on[r];
}

agent Robot {
  capabilities: move_robot, toggle_light;
  beliefs: full_state;
  intents: keep_lights_on(Human);
  constraints: none;
}

norms Safety {
  prohibition: Robot.move_robot(from, to) if to == ForbiddenRoom;
  obligation: light_on[location[Human]] == true;
  sanction: penalty += 1 if light_on[location[Human]] == false;
}

dynamics {
  joint_action {
    Robot.move_robot -> cost 1;
    Robot.toggle_light -> cost 1;
  }
  reward {
    Robot: keep_lights_on(location[Human]);
  }
}
```

For simplicity, we assume:

- The set of rooms is $\mathrm{Inst}(\mathrm{Room}) = \{r_1, r_2\}$.
- There is exactly one human and one robot: $\mathrm{Inst}(\mathrm{Human}) = \{h\}$, $\mathrm{Inst}(\mathrm{Robot}) = \{rob\}$.
- The constant `ForbiddenRoom` is r_2.

State Space

The state variables are:

- `location`: $(\mathrm{Human} \mid \mathrm{Robot}) \to \mathrm{Room}$.
- `light_on`: $\mathrm{Room} \to \mathrm{bool}$.

Concretely:

$$\text{location}: \{h, rob\} \rightarrow \{r_1, r_2\},$$
$$\text{light\_on}: \{r_1, r_2\} \rightarrow \{\text{true}, \text{false}\}.$$

Thus, the state space S has:

- $2^2 = 4$ choices for `location` (each of h and rob in r_1 or r_2);
- $2^2 = 4$ choices for `light_on`.

Hence $|S| = 16$ possible states.

We can denote a state as a tuple:

$$s = (\ell_h, \ell_{rob}, \lambda_1, \lambda_2),$$

where each $\ell_* \in \{r_1, r_2\}$ and each $\lambda_i \in \{\text{true}, \text{false}\}$ corresponds to light_on$[r_i]$.

Action Space

The robot has two capability schemas:

- `move_robot(from, to)` with $from, to \in \{r_1, r_2\}$.
- `toggle_light(r)` with $r \in \{r_1, r_2\}$.

Ignoring preconditions, the *candidate* action space is:

$$A_{\text{cand}} = \{\text{move\_robot}(r_i, r_j) \mid i, j \in \{1,2\}\} \\ \cup \{\text{toggle\_light}(r_i) \mid i \in \{1,2\}\}.$$

We can include a distinguished no-op action `idle` if desired.

Norms and Norm-Gated Actions

The norms declare:

- A `prohibition`: the robot may not move into `ForbiddenRoom` (r_2).
- An `obligation`: ensure the light is on in the human's current room.
- A `sanction`: penalty if the obligation is violated.

For the prohibition, the normative feasibility predicate $\phi_{\text{norm}}(s, a)$ can be defined as:

$$\phi_{\text{norm}}(s, a) = \begin{cases} \text{False}, & \text{if } a = \text{move\_robot}(r_1, r_2) \text{ or } a = \text{move\_robot}(r_2, r_2); \\ \text{True}, & \text{otherwise.} \end{cases}$$

Thus the norm-permissible set at any state s is:

$$A_N(s) = \{a \in A_{\text{cand}} \mid \phi_{\text{norm}}(s, a) = \text{True}\}.$$

The obligation "`light_on[location[Human]] == true`" is represented as a state property. It does not directly restrict $A_N(s)$ but induces a penalty if violated:

$$g_{\text{light}}(s,a) = \begin{cases} 1, & \text{if light\_on[location[Human]] == false;} \\ 0, & \text{otherwise.} \end{cases}$$

Dynamics and Utility

We assume deterministic environment dynamics for simplicity:

- `move_robot(from,to)` updates $\ell_{rob} := to$.
- `toggle_light`(r) updates $\lambda_r := \neg\lambda_r$.

The human and other aspects are static in this tiny example.

The robot's intent `keep_lights_on(Human)` can be modeled as a utility:

$$\widetilde{U}_{rob}(s,a) = \begin{cases} 0, & \text{if light\_on[location[Human]] == true;} \\ -c, & \text{if light\_on[location[Human]] == false,} \end{cases}$$

for some constant $c > 0$, possibly combined with a small action cost (from the `joint_action` costs).

Including the sanction as an additional penalty leads to an effective utility

$$\widetilde{U}'_{rob}(s,a) = \widetilde{U}_{rob}(s,a) - \lambda\, g_{\text{light}}(s,a),$$

for a multiplier $\lambda \geq 0$.

Example Trajectory

Suppose initially:

$$s_0 = (\ell_h = r_1, \ell_{rob} = r_1, \lambda_1 = \text{false}, \lambda_2 = \text{false}).$$

At $t = 0$, the robot can choose among actions in $A_N(s_0)$:

- It *cannot* choose `move_robot(r_1, r_2)` by prohibition.
- It *can* choose `toggle_light(r_1)` or `idle`.

If it selects `toggle_light(r_1)`, then:

$$s_1 = (\ell_h = r_1, \ell_{rob} = r_1, \lambda_1 = \text{true}, \lambda_2 = \text{false}),$$

and the obligation is satisfied, yielding higher utility and no sanction.

If it selects `idle`, the obligation is violated; the robot incurs penalty and negative utility. Over time, an optimal policy would keep the human's room lit, subject to prohibition on entering r_2.

This fully specifies S, A_N, T, and $\widetilde{U}'_{rob}$ for the tiny model.

Integration with Foundation Models and Edge Systems

ErisML is solver-agnostic: it does not dictate whether decisions come from symbolic planners, RL agents, or foundation models. Instead, ErisML standardizes state and constraints and wraps the FM within a policy gate.

A typical loop:

1. *Sensing:* raw sensor data are processed by encoders into structured features assigned to ErisML state variables.
2. *FM Reasoning:* an FM receives a task description and state summary, and proposes tool calls or high-level actions.
3. *Norm Gate:* the ErisML runtime instantiates candidate actions, filters them through $A_N(s)$, and passes the remainder to controllers.
4. *Execution and Logging:* chosen actions are executed; the runtime logs state, actions, norms, and any blocked moves for NVR/ADV estimation.

High-impact actions can be subject to additional verification or human confirmation.

Toolchain and Execution Architecture

A reference ErisML toolchain comprises:

- **Parser and Validator:** checks syntax and types.
- **Semantic Checker:** detects unreachable actions, contradictory norms, and ill-typed dynamics.
- **Compiler:** maps ErisML models to PDDL, RL environments, game-description languages, or domain-specific simulators.
- **Verifier / Policy Gate:** enforces norms at plan time and runtime; tracks violations.
- **Monitor / Logger:** maintains audit logs with cryptographic attestation hooks.
- **IDE and Visualization (future work):** supports graphical editing and debugging.

Evaluation and Scenario-Grounded Testing

ErisML supports scenario-grounded evaluation by using ErisML model instances as reusable test artifacts. Evaluation can cover:

- Capability coverage (tasks, contexts, modalities).
- Safety metrics (NVR, harm proxies).
- Longitudinal metrics (ADV, stability–plasticity).
- Human-centered metrics (trust, legibility, recoverability).

Red teaming is implemented by constructing adversarial scenarios and norms, then checking whether agents attempt disallowed actions or exhibit high NVR under stress.

Applications

We briefly highlight three application domains.

Healthcare at Home

An ErisML model includes patients, rooms, devices, and vitals as state; agents such as home assistants and clinicians; and norms around consent, escalation, and privacy. ErisML constrains access to sensitive data and encodes when alerts must be fired.

This scenario models:

- **Rooms**, **patients**, **vital sensors**, **medication devices**
- **AI home assistant** (agent) collaborating with **remote clinician**
- **Norms:** consent requirements, escalation, privacy boundaries
- **Rules:** when to raise an alert, when data access is allowed/forbidden

---

ErisML Example — Healthcare Home Monitoring

```
environment HomeCare {
  objects: Patient, Room, Device;

  state:
    location: (Patient | Agent) -> Room;
    heart_rate: Patient -> int;
    med_dispenser_locked: Device -> bool;
    has_consent: Patient -> bool;
    alert_active: bool;

  dynamics:
    update_vitals(p: Patient, new_hr: int)
      updates heart_rate[p] = new_hr;
    move_agent(a: Agent, from: Room, to: Room)
      updates location[a] = to;

    trigger_alert()
      updates alert_active = true;
}

agent HomeAssistant {
  capabilities: update_vitals, move_agent, trigger_alert;
  beliefs: full_state;
  intents: ensure_patient_safety(Patient), minimize_false_alerts();
}

agent RemoteClinician {
  capabilities: trigger_alert;
  beliefs: summary_only;
  intents: intervene_when_needed(Patient);
}

norms HomeCareNorms {

  # Privacy / consent
  prohibition: access(heart_rate[p]) unless has_consent[p] == true;

  # Escalation: if vitals cross threshold, alert MUST fire
  obligation: if heart_rate[p] > 150 then trigger_alert();

  # Alert rate control: Assistant may NOT trigger an alert more
  # than once within a cooldown window.
  prohibition: trigger_alert() if recent_alert_within(300s);

  # Devices: medication dispenser cannot unlock unless alert_active is
true
  prohibition: unlock_med_dispenser(d) unless alert_active == true;

  sanction: penalty += 10 if heart_rate[p] > 150 AND alert_active ==
false;
}

dynamics {
  joint_action {
    HomeAssistant.update_vitals -> cost 1;
    HomeAssistant.trigger_alert -> cost 3;
    RemoteClinician.trigger_alert -> cost 2;
  }
  reward {
    HomeAssistant: ensure_patient_safety(p) - minimize_false_alerts();
    RemoteClinician: intervene_when_needed(p);
```

```
      }
    }
```

Smart Campus Mobility

ErisML can model buildings, roads, vehicles, and pedestrians; agents controlling fleets; and norms such as right-of-way rules and speed limits, plus emergency overrides during evacuations.

This scenario models:

- Buildings, pathways, intersections, shuttles, delivery bots
- Speed limits, right-of-way rules, geofencing
- Special emergency modes enabling override of normal mobility rules
- Conflicts between efficiency and safety

ErisML Example — Campus Mobility Coordination

```
environment Campus {
  objects: Building, RoadSegment, Pedestrian, Shuttle, Bot;

  state:
    shuttle_location: Shuttle -> RoadSegment;
    bot_location: Bot -> RoadSegment;
    pedestrian_location: Pedestrian -> RoadSegment;
    congestion: RoadSegment -> int;
    emergency_mode: bool;

  dynamics:
    move_shuttle(s: Shuttle, to: RoadSegment)
      updates shuttle_location[s] = to;

    move_bot(b: Bot, to: RoadSegment)
      updates bot_location[b] = to;

    move_pedestrian(p: Pedestrian, to: RoadSegment)
      updates pedestrian_location[p] = to;

    set_emergency_mode(flag: bool)
      updates emergency_mode = flag;
}

agent MobilityOrchestrator {
  capabilities: move_shuttle, move_bot, set_emergency_mode;
  beliefs: full_state;
  intents: minimize_congestion(), ensure_pedestrian_safety();
}
```

```
agent ShuttleDriver (as an autonomous agent) {
  capabilities: move_shuttle;
  beliefs: near_local_state;
  intents: efficient_routes();
}

norms CampusRules {

  # Pedestrian priority: vehicles yield at crosswalks
  prohibition: move_shuttle(s,to)
               if pedestrian_in_crosswalk(to) AND emergency_mode == false;

  prohibition: move_bot(b,to)
               if pedestrian_in_crosswalk(to) AND emergency_mode == false;

  # Speed limits encoded as state conditions
  prohibition: move_shuttle(s,to)
               if speed_limit_exceeded(s,to);

  # Emergency override
  permission: move_shuttle(s,to)
              if emergency_mode == true;

  # Sanctions for conflict with pedestrians
  sanction: penalty += 20 if proximity_conflict(s,p);
}

dynamics {
  joint_action {
    MobilityOrchestrator.move_shuttle -> cost 2;
    MobilityOrchestrator.move_bot -> cost 1;
  }
  reward {
    MobilityOrchestrator: minimize_congestion() + ensure_pedestrian_safety();
  }
}
```

Industrial Maintenance

Machines, sensors, workers, and physical zones are part of the environment. Norms encode lockout–tagout procedures and restricted zones. Maintenance policies must satisfy these norms while optimizing uptime and cost.

This scenario models:

- Machines, sensors, workers, zones

- Lockout–tagout (LOTO) safety rules
- Hazardous zones
- Maintenance actions requiring multi-step norms and verifications
- Safety–uptime optimization
- Alarms and shutdown behavior

ErisML Example — Industrial Maintenance Safety

```
environment Factory {
  objects: Machine, Worker, Zone, Sensor;

  state:
    machine_running: Machine -> bool;
    sensor_value: Sensor -> int;
    worker_location: Worker -> Zone;
    zone_restricted: Zone -> bool;
    machine_locked_out: Machine -> bool;

  dynamics:
    read_sensor(s: Sensor, value: int)
      updates sensor_value[s] = value;

    move_worker(w: Worker, to: Zone)
      updates worker_location[w] = to;

    start_machine(m: Machine)
      updates machine_running[m] = true;

    stop_machine(m: Machine)
      updates machine_running[m] = false;

    lockout_machine(m: Machine)
      updates machine_locked_out[m] = true;

    release_lockout(m: Machine)
      updates machine_locked_out[m] = false;
}

agent MaintenanceBot {
  capabilities: read_sensor, move_worker, lockout_machine, release_lockout;
  beliefs: partial_state;
  intents: optimize_uptime(), follow_safety_protocols();
}

agent HumanTechnician {
  capabilities: lockout_machine, release_lockout, start_machine, stop_machine;
  beliefs: local_observation;
  intents: complete_repairs();
}

norms LOTO {
  # LOTO rule: worker entering a restricted zone requires machine lockout
  prohibition: move_worker(w,to)
              if zone_restricted[to] == true AND
machine_locked_out[affected_machine(to)] == false;

  # Machine cannot start while someone is inside hazardous zone
  prohibition: start_machine(m)
              if worker_in_hazard_zone(m);

  # Sanction for safety-critical violations
  sanction: penalty += 50 if proximity_violation(w,m);
}

dynamics {
  joint_action {
    MaintenanceBot.lockout_machine -> cost 5;
    HumanTechnician.start_machine -> cost 1;
  }
  reward {
    MaintenanceBot: optimize_uptime() - follow_safety_protocols();
    HumanTechnician: complete_repairs();
  }
}
```

Related Work

ErisML relates to planning languages (e.g., PDDL), decision-theoretic models (MDPs, POMDPs), agent-based modeling platforms, normative/deontic logics, and governance frameworks for FMs (e.g., Constitutional AI). Unlike these, ErisML aims to be a unified, executable specification language that integrates environment models, agents, and norms for solver-agnostic use and governance.

Limitations and Future Work

ErisML currently assumes discrete time and finite object sets; extending the semantics to continuous and hybrid systems is important for robotics and power systems. Compositionality—how to safely compose ErisML models—is not fully formalized. Tooling remains prototype-level; industrial deployments will require robust IDEs, conformance tests, and integration with governance stacks (e.g., democratically governed ethical modules). Finally, uncertainty modeling and robust optimization under distribution shift warrant deeper treatment.

Conclusion

We have presented ErisML, a formal modeling language for FM-enabled agents in pervasive computing environments. By unifying environment specification, agency, intent, norms, and dynamics, ErisML reduces fragmentation across planners, RL systems, and simulators, clarifies the normative envelopes within which agents may operate, and provides artifacts that regulators and auditors can inspect. A formal grammar and worked example demonstrate that ErisML can be given precise syntax and semantics while remaining accessible. We hope ErisML can serve as a substrate for the next generation of ambient, governed AI systems.