An Agent-Oriented Model-Driven Development Process for Cyber-Physical Systems

Claudio Navarro[1,2], Lorenzo Devia[3], Jose E. Labra Gayo[2], Carlos Cares[1]

[1]Computer Science and Informatics Department,
University of La Frontera, Temuco, La Araucanía, Chile

[2]Departamento de Informática, Universidad de Oviedo, Oviedo, Spain

[3]Graduate of Civil Engineering in Computer Science,
University of La Frontera, Temuco, La Araucanía, Chile

claudio.navarro@ufrontera.cl, l.devia02@ufromail.cl,
labra@uniovi.es, carlos.cares@ceisufro.cl

Abstract. *Cyber-Physical Systems (CPS) integrate distributed software and hardware, requiring systematic engineering approaches. This paper presents a Model-Driven Development (MDD) process that spans the entire CPS lifecycle, from conceptual modeling (CIM) to functional implementation (Code). By leveraging agent orientation, the approach simplifies system structuring and enables semi-automated transformations through a domain-specific language (DSL). A proof of concept validates the process in a greenhouse automation scenario, demonstrating that the generated software functions as expected on real hardware. The results confirm the feasibility of this end-to-end MDD workflow for CPS development.*

1. Introduction

Cyber-Physical Systems (CPS) integrate computational and physical processes across distributed components, leading to inherent development complexity [Cares et al. 2019, de C Henshaw 2016]. Structured methodologies are required to manage this complexity and align system implementation with high-level objectives [Sjöberg et al. 2023]. Agent orientation has been proposed as a suitable paradigm for structuring CPS, as it allows modeling components as autonomous entities that interact and adapt to their environment [Cares et al. 2019]. However, despite its potential, integrating agent orientation into CPS engineering remains an open challenge. Model-Driven Development (MDD) provides a systematic approach to software and system design through successive transformations from abstract models to concrete implementations [Atkinson and Kühne 2005, Selic 2003, Bézivin 2005]. While previous works have explored the application of MDD in CPS [Staroletov et al. 2019, Cares et al. 2022a, Jiang et al. 2018, Gräßler et al. 2021], a fully integrated process that spans from conceptual modeling to execution in real hardware has not been fully established [Liu et al. 2020]. An extensive systematic literature review on Model-Driven Development for Cyber-Physical Systems, as reported in [Mohamed et al. 2021], identified several underexplored areas, including requirements, socio-technical aspects, and the software of cyber components such as sensors and actuators. The study also noted limited coverage of the full development cycle, particularly model-to-model and model-to-code transformations, as well as the availability of technological solutions. We address these limitations as an integrated set to be covered. This

work presents a complete MDD process for CPS, structured in four stages: Computation-Independent Modeling (CIM), Platform-Independent Modeling (PIM), Platform-Specific Modeling (PSM), and Code. The approach leverages agent orientation in the CIM phase using iStar 2.0, which is then systematically transformed through a domain-specific language (DSL) into concrete CPS constructs. To support these transformations, we developed custom tools integrated with diagrams.net, enabling semi-automated model-to-model and model-to-code conversions. The feasibility of the proposed workflow is validated through a proof of concept in greenhouse automation. The generated software is deployed on Arduino-based computing units, verifying sensor-actuator interaction and message exchange via MQTT[1]. Results confirm that the process effectively supports the structured development of CPS while maintaining traceability and reducing manual intervention.

2. Theoretical Background

2.1. Cyber-physical Systems

Cyber-physical systems (CPS) offer solutions to manage intricate physical processes through monitoring and control. These systems utilize both software and hardware components that can detect variables from the physical world and respond accordingly. Owing to the distributed nature of their software and hardware across multiple devices, these systems inherently carry complexity [Cares et al. 2019]. CPS, conceptualized in the early 21st century, has been pivotal to the advent of Industry 4.0 [Lee et al. 2015]. The affordability of electro-mechanical devices has been a key factor in the widespread adoption of Cyber-Physical Systems (CPS). This trend is further driven by their extensive interconnectivity and the economic accessibility of dependable, high-quality sensors and microcontrollers [Norris 2015]. Yet, even with these technological advancements, methodologies and modeling techniques tailored for cyber-physical engineering remain in their nascent stages [Oudina and Derdour 2023, Liu et al. 2017, Cares et al. 2019, Cares et al. 2022b].

2.2. Agent Orientation Overview

Agent orientation is an approach in software engineering that emphasizes the importance of autonomous agents as primary entities in modeling and designing software systems. At its core, this paradigm recognizes agents as autonomous units that interact, perceive, and act based on their set objectives and the environment in which they operate. The foundation for agent-oriented methodologies can be traced back to significant works such as the iStar framework, introduced in 1995. This influential framework has since shaped requirements standards, further agent-oriented methodologies, modeling paradigms, and even the development of security and normative formalisms [Cares et al. 2011]. One of the notable methodologies influenced by the iStar framework is Tropos [Bresciani et al. 2004]. Central to Tropos are concepts like actors, which can further be categorized into agents, roles, and positions, and their objectives captured through goals, *softgoals*, resources, and tasks. Notably, goals can be further detailed or contribute to other goals, while *softgoals* delineate quality goals and non-functional requirements. An integral part of both iStar and Tropos is the representation of agents' interdependencies, viewed through the

[1]Message Queuing Telemetry Transport (MQTT) is a lightweight messaging protocol widely used for communication in IoT and CPS environments.

lens of their objectives and depicted in dependency diagrams. Nevertheless, the landscape of agent-oriented software engineering has also been enriched by other methodologies. GAIA [Wooldridge et al. 2000], O-MASE [Padgham and Winikoff 2005], and PASSI [Burrafato and Cossentino 2002] have each brought significant insights and techniques to the table, contributing to a comprehensive understanding and practice of agent-oriented development.

2.3. Model-Driven Development

In software engineering, models serve as abstract representations that highlight relevant aspects of a system while omitting unnecessary details [Pidd 2003]. They enhance understanding, document system components, and improve team communication, supporting informed design decisions [Selic 2003]. The Model-Driven Development (MDD) paradigm emerged in the early 2000s [Selic 2003, Bézivin 2005], proposing that software can be developed through successive computer-assisted transformations, starting from abstract models and progressing through increasingly detailed representations until executable code is generated. MDD shifts the focus from code-centric to model-based development, where models are the primary artifacts [Staron 2006]. This approach separates system logic from platform-specific implementation, allowing for automated transformations between abstraction levels [Atkinson and Kühne 2005, Bézivin 2005].

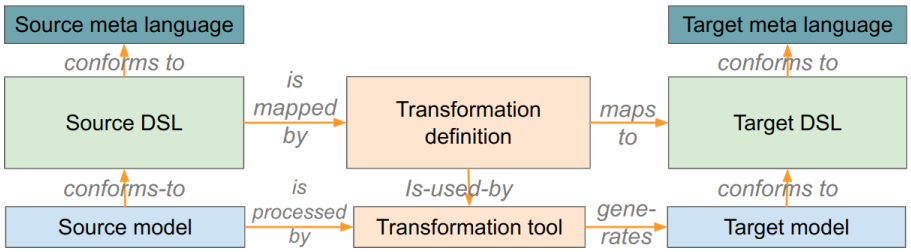

Figure 1. Model-to-model transformation in MDD.

As depicted in Figure 1 (adapted from [Kleppe et al. 2003]), these transformations are crucial for automating model evolution. A transformation definition specifies mappings from source to target models, and transformation tools execute these mappings to generate the target model. This process can be fully automated when the models contain all required information, or semi-automated when additional user input is necessary. Source and target models may use the same or different modeling languages, depending on the metamodels involved. MDD does not impose a fixed number of abstraction levels or specific standards, providing flexibility to adapt modeling approaches to the problem domain. It often leverages domain-specific languages (DSLs) to streamline modeling and transformations at each phase [Kleppe et al. 2003]. MDD approaches underpin modern low-code approaches, not aiming to eliminate manual coding entirely, but rather focusing on automating development tasks to reduce coding effort. Compared to traditional software development, MDD improves precision and efficiency, particularly in complex systems [Pastor et al. 2016].

2.4. Towards an Agent-Oriented MDD approach for CPS development

This work conceptualizes cyber-physical systems (CPS) as a network of cyber-physical components (CPCs) with sensing, actuation, communication, and computational capabil-

ities. Their computational capacity enables local processing, decision-making, and interaction within the system, serving as the execution environment for the software generated through the MDD process. The proposed CPS development process follows a semi-automatic MDD approach based on successive, well-defined stages. Its goal is to systematically capture design definitions, leading to the semi-automatic generation of code. The process considers three key modeling levels: the computation-independent model (CIM), which describes the system domain without technological details; the platform-independent model (PIM), which represents the system design independent of platform constraints; and the platform-specific model (PSM), which details the implementation for a specific platform. Given the conceptual framework reviewed and the nature of CPCs, the CIM level is approached from an agent-oriented perspective using the iStar 2.0 modeling language [Dalpiaz et al. 2016]. The PIM level relies on a domain-specific language (DSL) designed to provide an intermediate description of the CPS functionality, structure, and behavior, independent of specific technologies. Finally, the PSM and Code stages are devoted to code generation, configuration, and customization, as depicted in Figure 2.

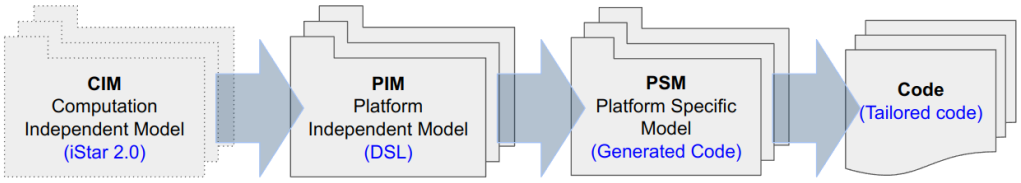

Figure 2. MDD process from Agents to CPS.

3. MDD for CPS: From Agents to Cyber-physical Systems

The following sections describe the CIM, PIM, PSM, and Code phases, detailing the modeling languages, their constructs, the transformations applied, and their role in CPS development.

3.1. Computation-Independent Models (CIM) for CPS

The Computation-Independent Model (CIM) phase defines the CPS at a high level, capturing domain requirements without technological details. This work adopts an agent-oriented approach, conceptualizing CPS as a network of autonomous cyber-physical components (CPCs) that exhibit proactivity, reactivity, and interactivity [Cares et al. 2011]. These CPCs include both physical entities—such as sensors, actuators, and communication modules—and software processes like control algorithms and resource management, bridging the cyber and physical domains. To support modeling in this phase, we use iStar 2.0, leveraging its constructs to define the structural and behavioral aspects of CPS. We developed an iStar 2.0 library for diagrams.net, shown in Figure 3, which facilitates practical modeling. The following sections describe each iStar 2.0 construct, its role in CPS modeling, and its dependency relationships.

Actor, Agent, Role: In iStar, the concept of Actor represents the primary autonomous entities in this paradigm. An actor can be further specified as an Agent, which refers to an autonomous and proactive entity, or as a Role, which abstracts the entity's function within the system without considering its individuality. In the context of CPS, actors are not limited to high-level abstractions of physical hardware components, such as

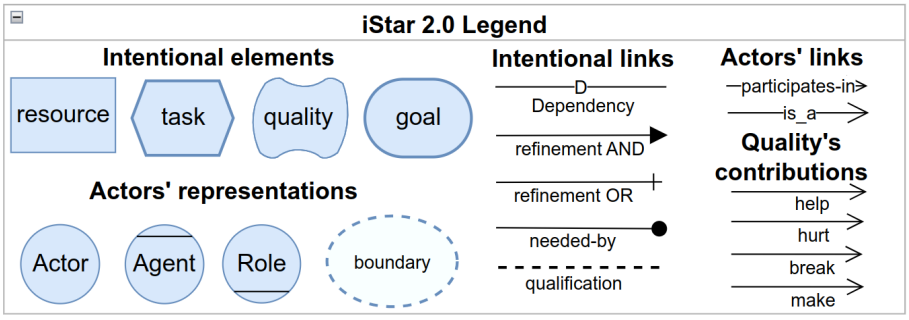

Figure 3. iStar 2.0 legend implemented as a library for diagrams.net.

sensors and actuators, or cyber entities, such as algorithms. They also provide a broader scope for comprehensive requirements modeling, encompassing stakeholders, environmental factors, and even user intentions. This allows for a more holistic representation of the system and its interactions.

Goals: These delineate the objectives an actor seeks to achieve. It is important to note that this element is not prescriptive, meaning that the specific means of achieving the goal are not constrained. For CPS, goals can represent desired states or outcomes, such as maintaining a specific speed or achieving a particular energy consumption level.

Softgoals: Representing non-functional requirements or quality attributes, softgoals in CPS might entail parameters like system resilience, security, or performance.

Resources: They can be tangible or intangible assets used by actors. This could relate to available bandwidth, power sources, computing capacity, or data storage in a CPS environment.

Tasks: Refers to the specific actions actors undertake to achieve their goals. Unlike goals, tasks are prescriptive and must explicitly define how they should be carried out. In the realm of CPS, tasks can encompass activities such as sensing, data acquisition, signal processing, or actuation within the physical environment.

Social dependencies: This construct captures the relations and dependencies between actors, crucial for the interconnected nature of CPS components. Figure 4 illustrates an iStar 2.0 dependency relationship, where the *Depender* agent relies on the *Dependee* agent to obtain the *dependum*. The *dependerElmt* explains *why* the *dependum* is needed, while the *dependeeElmt* specifies *how* it will be provided. In the diagram, both elements are represented as dashed-line boxes to denote that they can correspond to any intentional element in the model. The *dependum* itself can take different forms: a *Goal*, where the *Dependee* is responsible for achieving it with freedom in its execution; a *Quality* (*softgoal*), where the *Dependee* ensures sufficient compliance; a *Task*, which must be executed as prescribed; or a *Resource*, where the *Dependee* provides a specific asset.

Refinement, Needed-by, Contribution, and Qualification links: These links define how intentional elements interact within the model. *Refinement links* specify how goals and tasks are decomposed into more specific elements. *Needed-by links* indicate the resources required for task execution. *Qualification links* associate softgoals with the intentional elements they assess. Finally, *Contribution links* define how intentional elements impact softgoals, either positively (*help, make*) or negatively (*hurt, break*).

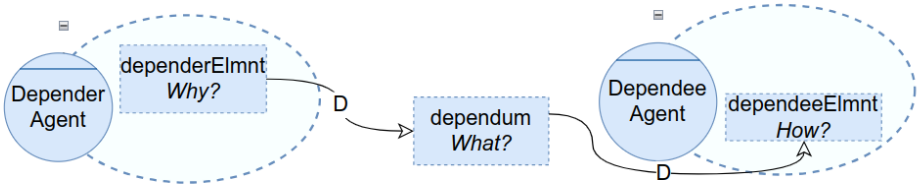

Figure 4. iStar 2.0 dependency relations.

As an example, Figure 5 illustrates a dependency relationship between an actuators and sensors, which can be considered part of a CPS for greenhouse automation. The model shows that both sensors and actuators have the primary *Goals* of measuring and maintaining the appropriate temperature, respectively. Additionally, both components aim to optimize resource usage in terms of energy consumption by analyzing battery usage and adjusting their operation mode accordingly.

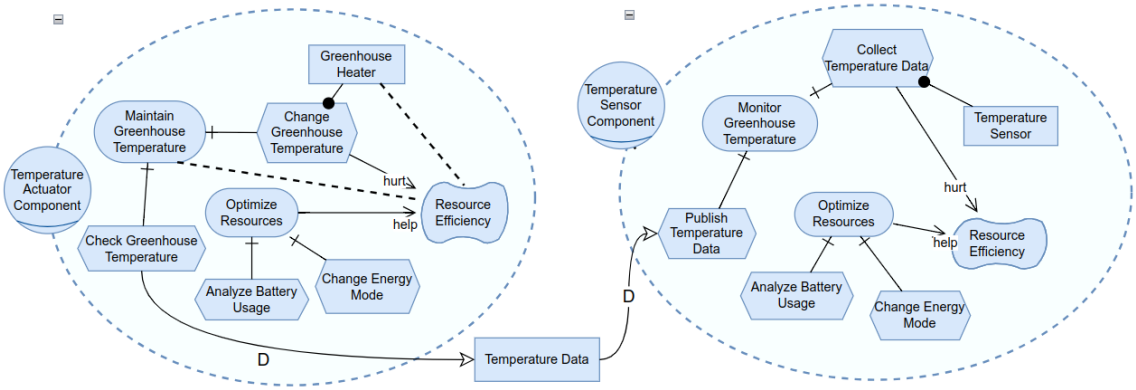

Figure 5. iStar 2.0 dependency relationship between a temperature actuator and a sensor.

It can also be observed that this process *helps* improve *Resource Efficiency*, whereas both collecting temperature data (by the sensor) and changing the temperature (via the heater) *hurt Resource Efficiency*. Furthermore, the model emphasizes *Resource Efficiency* for the actuator through *Qualification links*, ensuring that both the *Greenhouse Heater* and the *Goal* of maintaining the appropriate temperature in the greenhouse are aligned with this concern.

3.2. Platform-Independent Models (PIM) for CPS Modeling

The Platform-Independent Model (PIM) serves as an intermediate stage in the MDD approach, refining the CPS design without committing to specific technologies. To ensure a seamless transition from the agent-oriented CIM phase, we propose a domain-specific language (DSL) tailored for CPS. This DSL is designed to provide a straightforward and structured mapping of high-level agent-oriented constructs into CPS representations, facilitating model transformations and preserving traceability across MDD stages [Atkinson and Kühne 2005, Pastor et al. 2016]. Agent-oriented requirements from

CIM are refined into explicit CPS components at this stage. The transformation process ensures traceability by incorporating semantic annotations into PIM construct attributes, preserving information such as contribution and qualification relations with softgoals (*softgoal_qualification_array* and *softgoal_contribution_array*). These transformations are applied either automatically or with designer input when additional specifications are needed. To support accurate modeling, we developed a diagrams.net library ensuring a consistent representation of PIM constructs, attributes, and their relationships. Figure 6 presents the DSL legend, detailing the constructs and relationships used in this modeling phase.

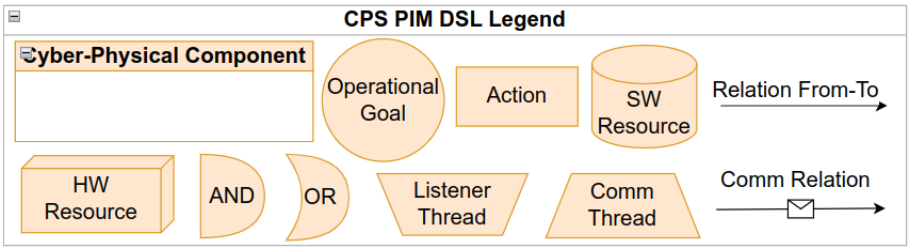

Figure 6. CPS PIM Domain specific language (DSL) legend.

Cyber-physical components (CPC): In PIM modeling, CPS are structured as a network of CPCs, each representing a computational unit that integrates sensing, actuation, and communication while interacting with other components. The focus is on their functional role rather than their specific implementation. CPCs serve as modular containers, encapsulating services such as data storage, computation, and decision-making in a technology-independent manner. This aligns with the component-based approach, where components abstract internal complexity and provide functionality through well-defined interfaces [Crnkovic 2005]. CPCs are derived from *Actors* in the agent-oriented CIM phase. While CIM actors can represent both cyber-physical and non-cyber-physical entities (e.g., stakeholders or external software systems), only explicitly cyber-physical actors are transformed into CPCs, as this work focuses on CPS development rather than general agent-based software. Identifying these actors requires designer interaction during transformation, ensuring correct classification. Once defined, traceability information from the original construct is preserved in the CPC attributes.

Operational Goals: An *Operational Goal* within a *CPC* (which is a computational unit) represents a process or thread aimed at achieving a specific goal from the previous CIM phase, without prescribing how it is fulfilled. It continuously evaluates its achievement based on inputs such as other *Operational Goals* or *Actions* that refine it, as well as internal criteria. This includes the attribute *checkInterval_in_milliseconds*, which defines its evaluation frequency. Additionally, operation modes can be specified through the attributes *operation_modes_enabled* (boolean) and *operation_modes_description* (text), allowing for discrete operational strategies when required [Navarro et al. 2024]. These attributes are defined through user interaction and apply only if discrete operation modes are needed.

Actions: An Action in a CPC represents a prescriptive procedure executed within the component, similar to a function in procedural programming. Unlike an Operational Goal, an Action is not autonomous but is triggered as needed. It returns a boolean value

indicating success or failure and may require input parameters and produce output parameters, both defined via user interaction in this phase, without specifying data types. Actions contribute to other Actions or Operational Goals via AND/OR operators, signaling success or failure. Additionally, like Operational Goals, they include the operationModes attribute to specify discrete operation modes when applicable.

SW Resources: An *SW Resource* in PIM represents a non-physical resource, specifically data required for the operation of other elements. It originates from a *Resource* in CIM, where resource types are not yet distinguished. During transformation, user input determines whether a *CIM Resource* should be classified as an *SW Resource* in PIM. If confirmed, the transformation proceeds by assigning a *data_structure* attribute, defined through an interaction with the designer. This definition includes field names and natural language descriptions, ensuring clarity without specifying data types.

HW Resources: A *HW Resource* in PIM represents a physical resource mapped from a *Resource* in CIM, requiring implementation within the corresponding *CPC*. This construct defines the hardware needed for proper system operation. Examples include servomotors, sensors, I/O ports, and battery packs, or complex devices like 3D printers, assumed to have necessary consumables. To determine whether a CIM *Resource* transforms into an *HW Resource* or *SW Resource*, user input is required.

AND, OR Operators: These operators in PIM originate from iStar 2.0 goal and task refinement relations in CIM. They are transformed while preserving their logical structure, now interpreted through the boolean evaluation of *Operational Goals* and *Actions*. In iStar, *Goals* and *Tasks* decompose into more specific elements via **AND** or **OR** refinements, a logic maintained in CPS-PIM: an *AND-refinement-operator* enforces that all related *Operational Goals* and *Actions* must succeed, while an *OR-refinement-operator* allows multiple alternatives, requiring at least one to be achieved. These operators ensure that the hierarchical breakdown of objectives from CIM is preserved in PIM, adapting it to a computational model where goal fulfillment is dynamically evaluated based on process execution.

Relation (from-to): In this PIM modeling language for CPS, each construct combination within a *Cyber-Physical Component* (CPC) has a single, unambiguous relation type. All CIM associations are mapped to this unified relation, where the constructs involved and the relation's direction define its nature.

Listener Thread, Comm Thread, and Comm Relation: Dependencies in CIM are transformed in PIM into structured message-based communication between *Cyber-Physical Components* (CPCs), as shown in Figure 7. The *Comm Thread* in the *Dependee CPC* gathers and transmits the *dependum* as a message via *Comm Relations* to the *Listener Thread* in the *Depender CPC*, which then distributes it internally.

Interactions with the designer allow defining the dependum's data structure (specifying fields but not data types, maintaining a technology-agnostic design). Additionally, designer interaction is required to specify the transmission frequency of the dependum, which is set through a time interval attribute in the *Comm Thread*. Other communication timing preferences—such as unacknowledged send, retransmission on demand, and single confirmation—can also be defined independently of technology at this stage as operation modes. Both the dependum's data structure and communication timing parameters must

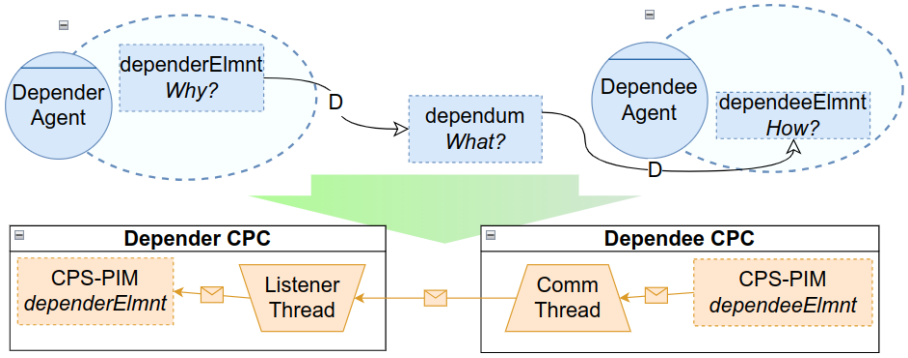

Figure 7. Transformation from CIM dependency relation to PIM communication process.

be adopted by the *Comm Thread* and its corresponding *Listener Thread*. If the *dependeeElmnt* or *dependerElmnt* in CIM is a *softgoal*, it is subsumed in PIM, meaning that no explicit softgoal-equivalent construct appears in the PIM phase, nor does any *Comm Relation* related to it. Instead, softgoal traceability is preserved within the corresponding *Comm Thread* and *Listener Thread* attributes, allowing this information to be leveraged in the PSM phase. This approach standardizes all dependencies in PIM, ensuring a uniform and flexible communication mechanism between CPCs.

Figure 8 illustrates the transformation of the dependency relationship depicted in Figure 5 into a PIM model. This transformation represents both the logical refinement relations between *Operational Goals* and *Actions* and the structured communication process facilitated by *Comm Threads* and *Listener Threads*.

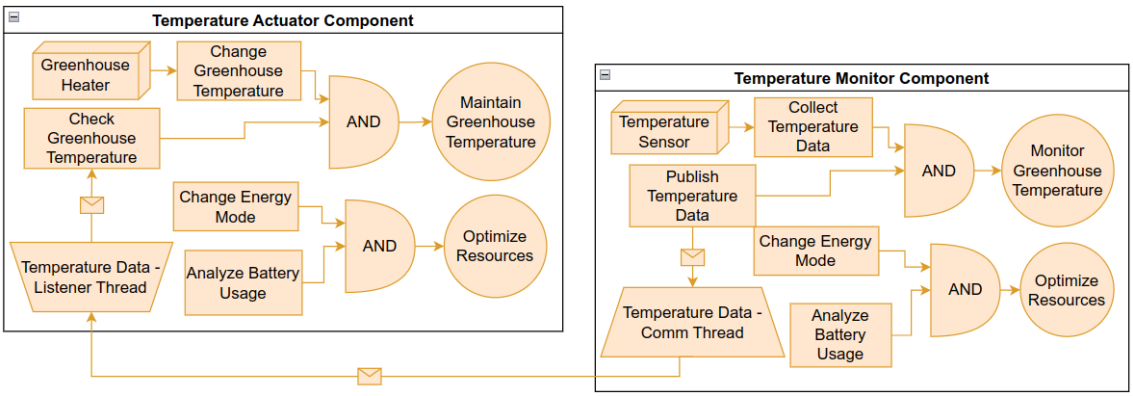

Figure 8. Example CPS PIM DSL: Communication process between a temperature actuator and a sensor.

3.3. Platform-Specific Models (PSM) for CPS Implementation

The Platform-Specific Model (PSM) phase converts PIM representations into functional, executable, or at least compilable code for the selected platform. This process requires interaction with the designer during the PIM-to-code transformation to define data types, such as input and output parameters for *Actions* (attributes *input_parameters* and *output_parameters*) and *dependum* contents to ensure compatibility. Softgoal traceability, contribution, and qualification relations from the CIM phase are included as comments

to enhance traceability and improve developer efficiency. Sections for manual modifications are also clearly marked. The following outlines both general and target-specific transformations, with Arduino MKR WiFi 1010 and MQTT as the selected platform.

Cyber-Physical Components (CPC): Mapped to selected platform computing units, i.e., Arduino MKR WiFi 1010.

Operational Goals: Transformed into execution threads using FreeRTOS_SAMD21[2] for multitasking, including timing preferences for goal evaluation. When using operation modes, a switch statement is included to clearly define the behavior of each mode.

Actions: Converted into platform-specific functions, with explicitly defined input/output types determined through interaction with the designer and based on available data types on the target platform. The PIM-to-code transformation also supports function operation modes through switch statements.

SW Resources: Mapped to concrete data structures with defined field types, including traceability indications via block comments in the code.

HW Resources: The PIM-to-code transformation includes block comments with relevant information from previous stages regarding the required hardware in the functions that use them.

Communication Threads and Relations: This transformation maps these PIM constructs to code, inheriting timing synchronization on both sides of the communication, and defining the message data structure (formerly the dependum). It is implemented with MQTT, using PubSubClient.h[3] for constrained environments.

AND/OR Operators: Translated into execution control structures ensuring logical conditions. Functions (when invoked) and threads (periodically) evaluate logical expressions to report their accomplishment status.

This systematic transformation process ensures alignment with the original design while adapting to the constraints of the target platform.

3.4. Code Phase: Final Configuration and Customization

The code generated in the PSM phase is functional—it compiles and runs on the target hardware without programming errors. However, the Code phase focuses on configuring and fine-tuning the generated code for deployment. Typical adjustments include specifying WiFi credentials (SSID and password), selecting hardware connection pins, and adding necessary libraries for specific hardware, such as the DHT11[4] sensor. Additionally, this phase allows programmers to refine operation modes: although structurally defined in the generated code, specific behaviors for each mode often require manual customization. These final adaptations ensure the software aligns with the deployment environment while preserving the MDD structure.

[2] `FreeRTOS_SAMD21` refers to the real-time operating system (FreeRTOS) port adapted for the SAMD21 microcontroller used in Arduino MKR WiFi 1010 boards, enabling multitasking capabilities.

[3] `PubSubClient.h` is an Arduino library that provides MQTT client functionality for embedded devices.

[4] DHT refers to a family of digital humidity and temperature sensors, with DHT11 and DHT22 being common models differing in accuracy and range.

3.5. Demonstrating the MDD Approach: Developing a Cyber-Physical System for Greenhouse Automation

To validate the feasibility of conducting an MDD process for developing a CPS under an agent-oriented approach, we implemented a proof of concept (PoC). This PoC demonstrates the applicability of the approach using a set of integrated tools within diagrams.net, including libraries for CIM and PIM modeling, a containerized XSLT-based web tool for designer interaction, and model-to-model and model-to-code transformations. The PoC is considered successful if the generated software enables cyber-physical components (CPCs) to interact as expected in a greenhouse automation scenario. The system is tested by executing the automatically generated code on Arduino MKR WiFi 1010 units, ensuring that sensors collect data, actuators respond to control commands, and communication between components occurs via MQTT. Success criteria include: Proper execution of the generated Arduino sketches, with manual fine-tuning in the Code stage, Correct sensor data acquisition and transmission, Consistent actuation based on received commands, and Communication logs confirming successful message exchange between CPCs. If any of these criteria are not met, debugging is performed to determine whether the issue stems from the generated code (indicating a limitation in the MDD process) or external hardware/configuration factors.

As part of the CIM modeling stage, various actors were identified and modeled using iStar 2.0. Non-cyber-physical actors, such as the *Administrator* and *Middleware platform* roles, were discarded. The roles corresponding to *Soil Moisture Monitor Component*, *Soil Moisture Actuator Component*, *Temperature Monitor Component*, and *Temperature Actuator Component* were transformed and carried forward through the CPS development stages. Figure 9 illustrates the MDD process from a method engineering perspective, showing its step-by-step workflow alongside diagrams.net modeling tools and the web transformation tool. The final output consists of compiled and configured code for each CPC, fine-tuned during the Code stage and integrated with the hardware, forming the complete CPS.

During the Code stage fine-tuning, minor manual adjustments were necessary to integrate specific hardware components. For example, we manually included the library and reading functions for a DHT22 humidity and temperature sensor. These adjustments were straightforward thanks to the automatically generated traceability comments and the explicit placeholders indicating where manual code additions should be made. Another case involved the MQTT communication for the actuator CPC, where inconsistent readings were observed. This issue was resolved by fine-tuning the memory allocation for the communication thread. After completing the MDD process for the greenhouse automation CPS, we verified that all proof-of-concept (PoC) success criteria were met.

All the assets developed to support this agent-oriented MDD for CPS approach are publicly available, as detailed in the *Artifact Availability* section (see Section 5). These include the diagrams.net libraries for CIM (iStar 2.0) and PIM (DSL) modeling, the corresponding models used in the proof-of-concept, the PSM-generated Arduino sketches refined during the Code stage, and the complete source code of the transformation tool—both frontend and backend, covering transformation definitions, code generation patterns, and model processing logic. Step-by-step usage guidelines and documentation, are also provided.

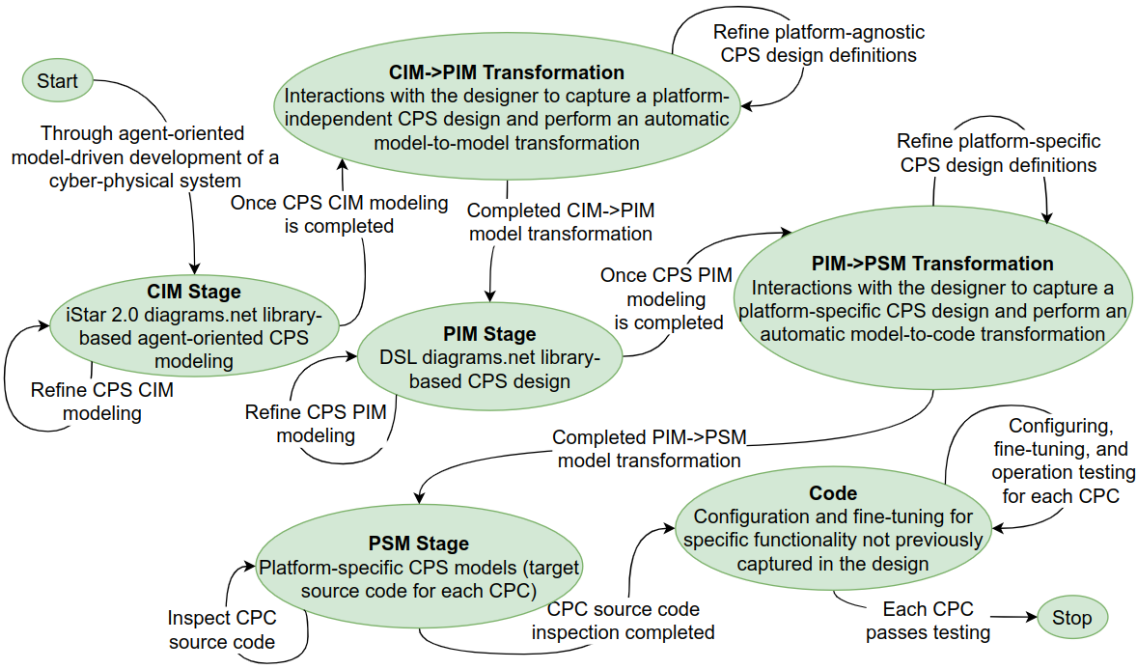

Figure 9. Agent-oriented CPS MDD process.

4. Results and Discussion

The proposed MDD process was successfully applied to CPS development, covering the full cycle from conceptual modeling (CIM) to implementation (Code). The proof of concept demonstrated that CPCs generated through this approach operated as expected, following their defined behaviors and interacting via MQTT according to the initial models. While the validation was conducted in a controlled environment, the results confirm that the process effectively supports CPS design and implementation. Agent orientation played a key role in the initial conceptualization of the CPS during the CIM phase. This paradigm shift enabled high-level abstraction units, simplifying analysis by decomposing the global problem into smaller, more manageable ones. These were then systematically transformed into concrete constructs in later stages, facilitating system implementation. The PoC was instrumental in verifying the process and identifying areas for improvement. As shown in Figure 9, each stage allows refinement through interactions with the web-based transformation tool. Designers provide technology-independent and technology-dependent information via web forms, whose complexity scales with model attributes. However, the current process flow does not allow revisiting previous phases without losing modifications from the current stage, which could be improved to increase design flexibility. A significant advantage was that the CPC software was tested on real hardware, not in a simulated environment. Comparatively, while previous studies have incorporated agent orientation into specific segments of CPS development [Ayala et al. 2019, Daun et al. 2021], this approach spans the entire lifecycle, from conceptualization to implementation. The initial phase leverages iStar 2.0 without specialized CPS extensions, ensuring compatibility with standard iStar 2.0 models. This allows the CIM stage to produce iStar 2.0-compliant models, seamlessly integrable with external iStar 2.0-based processes. Likewise, external iStar 2.0 models can be easily incorporated into this process, maintaining interoperability within the MDD workflow.

5. Conclusion

This work presented an MDD-based approach for CPS development, integrating agent orientation to structure the transition from conceptual modeling (CIM) to implementation (Code). The proposed process was validated through a proof of concept in greenhouse automation, demonstrating that CPCs generated via model transformations operated as expected, interacting through MQTT and adhering to their predefined behaviors. This confirms the feasibility of using MDD to develop CPS from high-level abstractions to functional implementations. Agent orientation played a fundamental role in managing complexity during the CIM phase, allowing high-level abstraction units to facilitate structured decomposition. This contributed to a systematic refinement process across MDD stages, preserving design intent while progressively incorporating CPS implementation details. Unlike previous works that apply MDD partially or focus on isolated phases, this approach covers the entire CPS development cycle, from conceptualization to execution in real hardware. It combines automated model transformations with guided designer interaction, progressively incorporating design details through a semi-automatic process aligned with each development phase, ensuring that technology-independent and technology-specific decisions are captured at appropriate workflow stages. Future improvements include enhancing the navigability of the MDD workflow and refining the usability of transformation tools. Allowing designers to select specific technologies and communication protocols per CPC could provide greater flexibility. The early inclusion of commonly used hardware components constructs could further reduce manual intervention in the Code phase. Additionally, incorporating design patterns from the CIM level may further streamline the initial modeling process by providing reusable templates for frequently occurring component compositions, reducing the need to redefine common structures. Beyond these refinements, a key direction for future work is transitioning from this descriptive implementation towards a formal specification of the languages and transformation rules used in the MDD process. While the current approach effectively supports automation and has been validated in a proof of concept, formalizing the semantics of the proposed DSL and the transformation rules could further enhance the rigor, reproducibility, and interoperability of the process, facilitating its integration with existing MDD frameworks and broader adoption. Furthermore, an in-depth evaluation of this approach's impact on CPS development efficiency is needed. Initial observations suggest a reduction in development time, but a systematic assessment in practical settings is required to compare its effectiveness against traditional methods.

Artifact Availability

The software artifact and documentation supporting this paper are available at Zenodo: `https://doi.org/10.5281/zenodo.15086005`

References

Atkinson, C. and Kühne, T. (2005). A generalized notion of platforms for model-driven development. In *Model-driven software development*, pages 119–136. Springer.

Ayala, I., Amor, M., Horcas, J.-M., and Fuentes, L. (2019). A goal-driven software product line approach for evolving multi-agent systems in the internet of things. *Knowledge-Based Systems*, 184:104883.

Bézivin, J. (2005). On the unification power of models. *Software & Systems Modeling*, 4(2):171–188.

Bresciani, P., Perini, A., Giorgini, P., Giunchiglia, F., and Mylopoulos, J. (2004). Tropos: An agent-oriented software development methodology. *Autonomous Agents and Multi-Agent Systems*, 8:203–236.

Burrafato, P. and Cossentino, M. (2002). Designing a multi-agent solution for a bookstore with the passi methodology. *AOIS@ CAiSE*, 57.

Cares, C., Chebanyuk, O., and Navarro, C. (2022a). A model-driven approach for cyber-physical systems: a method engineering perspective. *Proceedings of the XVII-th International Conference on Software Engineering SoftEngine 2022*, page 14.

Cares, C., Franch, X., Quer, C., and Enric, M. (2011). A reference model for i*. In Yu, E., Giorgini, P., Maiden, N., and Mylopoulos, J., editors, *Social Modeling for Requirements Engineering*, pages 573–606. MIT Press.

Cares, C., Lühr, D., Mora, S., Navarro, C., Olivares, L., Sepúlveda, S., and Vidal, G. (2022b). Architecting autonomous underwater vehicles by adapting software product lines. In *Conference on Integrated Computer Technologies in Mechanical Engineering–Synergetic Engineering*, pages 719–730. Springer.

Cares, C., Sepúlveda, S., and Navarro, C. (2019). Agent-oriented engineering for cyber-physical systems. In *Information Technology and Systems: Proceedings of ICITS 2019*, pages 93–102. Springer.

Crnkovic, I. (2005). Component-based software engineering for embedded systems. In *27th international conference on Software engineering*, pages 712–713.

Dalpiaz, F., Franch, X., and Horkoff, J. (2016). istar 2.0 language guide. *arXiv preprint arXiv:1605.07767*.

Daun, M., Brings, J., Krajinski, L., Stenkova, V., and Bandyszak, T. (2021). A grl-compliant istar extension for collaborative cyber-physical systems. *Requirements Engineering*, 26(3):325–370.

de C Henshaw, M. J. (2016). Systems of systems, cyber-physical systems, the internet-of-things... whatever next? *Insight*, 19(3):51–54.

Gräßler, I., Wiechel, D., Roesmann, D., and Thiele, H. (2021). V-model based development of cyber-physical systems and cyber-physical production systems. *Procedia CIRP*, 100:253–258.

Jiang, Y., Song, H., Yang, Y., Liu, H., Gu, M., Guan, Y., Sun, J., and Sha, L. (2018). Dependable model-driven development of cps: From stateflow simulation to verified implementation. *ACM Transactions on Cyber-Physical Systems*, 3(1):1–31.

Kleppe, A. G., Warmer, J. B., Warmer, J., and Bast, W. (2003). *MDA explained: the model driven architecture: practice and promise*. Addison-Wesley Professional.

Lee, J., Bagheri, B., and Kao, H.-A. (2015). A cyber-physical systems architecture for industry 4.0-based manufacturing systems. *Manufacturing letters*, 3:18–23.

Liu, B., Zhang, Y.-r., Cao, X.-l., Liu, Y., Gu, B., and Wang, T.-x. (2020). A survey of model-driven techniques and tools for cyber-physical systems. *Frontiers of Information Technology & Electronic Engineering*, 21(11):1567–1590.

Liu, Y., Peng, Y., Wang, B., Yao, S., and Liu, Z. (2017). Review on cyber-physical systems. *IEEE/CAA Journal of Automatica Sinica*, 4(1):27–40.

Mohamed, M. A., Kardas, G., and Challenger, M. (2021). Model-driven engineering tools and languages for cyber-physical systems–a systematic literature review. *IEEE Access*, 9:48605–48630.

Navarro, C., Gayo, J. E. L., Jara, F. A. E., and Cares, C. (2024). Componentizing autonomous underwater vehicles by physical-running algorithms. *PeerJ Computer Science*, 10:e2305.

Norris, D. (2015). *The internet of things: Do-it-yourself at home projects for arduino, raspberry pi, and beaglebone black*. McGraw-Hill Education.

Oudina, Z. and Derdour, M. (2023). Toward modeling trust cyber-physical systems: A model-based system engineering method. *International Journal of Advanced Computer Science and Applications*, 14(7):441–452.

Padgham, L. and Winikoff, M. (2005). *Developing intelligent agent systems: A practical guide*. John Wiley & Sons.

Pastor, Ó., España, S., and Panach, J. I. (2016). Learning pros and cons of model-driven development in a practical teaching experience. In *International Conference on Conceptual Modeling*, pages 218–227. Springer.

Pidd, M. (2003). *Tools for Thinking: Modelling in Management Science*. John Wiley and Sons Ltd, 2nd edition.

Selic, B. (2003). The pragmatics of model-driven development. *IEEE software*, 20(5):19–25.

Sjöberg, P., Mendez, D., and Gorschek, T. (2023). Contemporary challenges when developing cyber-physical systems of systems-a case study. In *2023 IEEE/ACM 11th International Workshop on Software Engineering for Systems-of-Systems and Software Ecosystems (SESoS)*, pages 46–53. IEEE.

Staroletov, S., Shilov, N., Konyukhov, I., Zyubin, V., Liakh, T., Rozov, A., Shilov, I., Baar, T., and Schulte, H. (2019). Model-driven methods to design of reliable multiagent cyber-physical systems. In *CEUR workshop proceedings*, volume 2478, pages 74–91.

Staron, M. (2006). Adopting model driven software development in industry–a case study at two companies. In *International Conference on Model Driven Engineering Languages and Systems*, pages 57–72. Springer.

Wooldridge, M., Jennings, N. R., and Kinny, D. (2000). The gaia methodology for agent-oriented analysis and design. *Autonomous Agents and multi-agent systems*, 3:285–312.