BELLS UNIVERSITY OF TECHNOLOGY-NEW HORIZONS

Robotics MBD Development Project Report: Obstacle Avoidance Robot

Mechatronics Engineering.

Team Members:

2023/12542, Anamalu Malachi Okechukwu.

2023/12378, Arowosaye Emmanuel.

2023/12643, Akinwola Israel.

2023/12144, Arowolo Mubarak.

2023/12137, Asenuga morifeoluwa.

Table of Contents

1. Introduction & Project Overview

The landscape of modern engineering, particularly in the realm of robotics, is undergoing a profound transformation, moving away from conventional sequential development methods towards more integrated and efficient paradigms. This report delves into the development of an advanced Obstacle Avoidance Robot, a project meticulously executed under the auspices of the Robotics Model-Based Design (MBD) Development Project at Bells University of Technology - New Horizons. This initiative underscores a strategic pedagogical shift, guiding students from foundational manual design and programming practices towards a sophisticated, model-driven engineering approach characterized by automatic code generation. The essence of this transition lies in leveraging abstract models to define, simulate, verify, and ultimately implement complex robotic systems, thereby minimizing errors, accelerating development cycles, and enhancing overall system quality.

The overarching objective of this endeavor was to engineer, simulate, and deploy a fully functional obstacle avoidance robot by rigorously applying Model-Based Design methodologies. MBD, as a comprehensive development paradigm, offers a myriad of compelling advantages that significantly surpass the limitations of traditional, code-centric development. These include, but are not limited to: a substantially higher level of abstraction that allows engineers to focus on system behavior rather than low-level code specifics; unparalleled rapid prototyping capabilities through continuous simulation, enabling swift design iterations and validation; the groundbreaking ability of automatic code generation, which virtually eliminates manual coding errors and ensures direct traceability from design to implementation; a marked reduction in development errors by identifying and rectifying design flaws at the earliest possible stages of the lifecycle; and vastly improved documentation, as the models themselves serve as living, executable specifications of the system's architecture and behavior, providing clarity and consistency throughout the project's lifespan.

This comprehensive report is structured to systematically detail the streamlined MBD workflow adopted for this project. Each phase, from the initial project definition and meticulous requirements gathering to the intricate stages of system architecture modeling, algorithm development and exhaustive simulation, the pivotal automatic code generation, multi-tiered testing and validation, strategic optimization and robust deployment, and finally, meticulous documentation and comprehensive reporting, will be elucidated. The insights gleaned from this project are expected to serve as a testament to the efficacy of MBD in addressing contemporary challenges in robotics engineering and to provide a robust framework for future innovations in this dynamic field.

2. Team Summary

This challenging yet highly rewarding project was collaboratively undertaken by a dedicated student team within the esteemed Engineering Departments of Bells University of Technology. The project was conceived with the explicit expectation of delivering a complete, original, and deployable software solution for the obstacle avoidance robot, alongside its mechanical design. The foundational frameworks instrumental in this development were SolidWorks, utilized for the meticulous mechanical design of the robot's physical body, and the integrated power of MATLAB/Simulink, serving as the cornerstone for Model-Based Design, simulation, and control algorithm development.

The project guidelines, meticulously outlined by the faculty, stipulated precise expectations, a structured timeline, and mandated the utilization of collaborative tools, primarily GitHub, to ensure a cohesive and successful outcome. This project was strategically designed to build upon the students' prior theoretical and practical foundation in Electronics Digital Automation, providing a robust transition to the more advanced and industry-relevant Model-Based Design methodology. This progression represents a significant leap forward, emphasizing automatic code generation as the preferred method for implementing robotics systems, thereby departing from the more labor-intensive manual coding practices. The collaborative nature of the project, coupled with its emphasis on cutting-edge design methodologies, aimed to cultivate essential skills in problem-solving, model-based system design, and effective team-based project management, all vital attributes for future engineering professionals.

3. Project Workflow Overview (Based on MBD Methodology)

The successful development of the Obstacle Avoidance Robot was predicated upon strict adherence to the Model-Based Design (MBD) methodology. This systematic approach provided a streamlined, iterative workflow meticulously crafted to maximize efficiency, ensure unparalleled accuracy, and facilitate long-term maintainability of the robotic system. This paradigm marks a significant departure from the conventional manual design and intricate programming of electronic systems encountered in earlier academic endeavors, unequivocally favoring a model-driven development approach underpinned by the power of automatic code generation. The pivotal stages of this comprehensive workflow are articulated below, highlighting the sequential yet interconnected nature of MBD:

Project Definition & Requirements: This foundational phase initiated with the precise selection of the "Obstacle Avoidance Robot" as the project's central theme. Following this, a rigorous process of functional and technical requirements elicitation and documentation was undertaken. The emphasis during this stage was not merely on listing requirements, but critically, on conceptualizing how these requirements would be seamlessly translated into verifiable and executable models within the MBD environment. This early-stage focus on model-centric requirements is crucial for aligning design with intent from the outset.

System Architecture & Modeling: This phase represents the very heart of the MBD approach. Here, the robot's holistic architecture and its intricate behaviors were meticulously defined and graphically represented using sophisticated modeling tools such as Simulink for dynamic system behavior and control logic, and SolidWorks for the detailed mechanical design of the robot's physical form. The creation of interconnected blocks and signals, hierarchical subsystems, and state-based logic formed the bedrock of the system's executable blueprint.

Algorithm Development & Simulation: With the system architecture in place, the algorithms essential for obstacle detection, avoidance, and general navigation were developed not as traditional code, but as robust models within the Simulink environment. These algorithms were then subjected to exhaustive simulation campaigns. This iterative simulation process allowed for precise parameter optimization, thorough verification of model behavior under diverse conditions, and the early identification and rectification of logical flaws or performance bottlenecks, all within a safe, virtual setting.

Automatic Code Generation: This constitutes a truly transformative phase within the MBD workflow. Here, the meticulously designed and rigorously verified models were automatically translated into highly optimized, deployable executable code tailored for the robot's target embedded platform. This automated translation virtually eliminates the risk of manual coding errors, drastically reduces development time, and crucially, ensures direct and indisputable traceability between the high-level design models and the low-level implemented code.

Testing & Validation: Rigorous and multi-tiered testing was an indispensable component throughout the project lifecycle. Verification efforts spanned various levels, including Model-in-the-Loop (MIL), Software-in-the-Loop

(SIL), Processor-in-the-Loop (PIL), and ultimately, Hardware-in-the-Loop (HIL) testing. Each stage was designed to systematically validate the system's performance against its defined requirements, progressively moving from purely virtual simulations to the interaction with actual hardware components, thereby building confidence in the system's robustness and reliability.

Optimization & Deployment: Based on the invaluable insights and data accrued from the comprehensive testing phases, the models underwent iterative refinement and optimization. Subsequently, optimized code was regenerated to enhance performance characteristics such as execution speed, memory footprint, or power efficiency. This refined code was then meticulously integrated with the robot's physical hardware, culminating in rigorous field testing under real-world conditions to validate the robot's operational efficacy and stability in its intended environment.

Documentation & Reporting: A commitment to comprehensive documentation and structured reporting was maintained throughout the entire development cycle. This included detailed documentation of all created models, the automatically generated code, and the overarching project narrative. This emphasis on robust documentation ensures knowledge retention, facilitates future maintenance and upgrades, and provides a clear audit trail of design decisions and implementation details.

4. Detailed Implementation Guide

This section provides an in-depth exposition of each crucial phase of the project's implementation. It meticulously details the specific activities undertaken, the cutting-edge tools employed, and the precise methodologies applied during the development of the Obstacle Avoidance Robot, offering a granular view of the MBD approach in action.

4.1. Project Definition and Requirements Gathering

The genesis of the project commenced with a deliberate and strategic selection of the "Obstacle Avoidance Robot" as the central theme, a choice that inherently allowed for a natural progression from the students' foundational knowledge acquired in Electronics Digital Automation. To ensure a clear trajectory and measurable success, explicit and quantifiable objectives were rigorously defined, specifically tailored to demonstrate the advancements achieved through the adoption of the MBD paradigm.

The process of requirements gathering was conducted with meticulous attention to detail, leading to the thorough documentation of both functional and technical specifications. A distinguishing characteristic of this phase, aligned with MBD principles, was the pronounced focus on conceptualizing how these elicited requirements would seamlessly translate into verifiable and executable models, rather than prematurely considering direct code implementation. Key requirements that guided the robot's design and functionality included:

Obstacle Detection: A paramount functional requirement dictating that the robot must possess the capability to accurately sense and identify the presence of obstacles within a predetermined operational range, regardless of their material or size within defined limits. This necessitates robust sensor integration and reliable data processing.

Avoidance Maneuver: Upon the positive detection of an obstacle, the robot is critically required to execute a precise and effective maneuver to unequivocally prevent collision. This involves sophisticated decision-making logic to choose the optimal avoidance path (e.g., turning left, right, or reversing) based on environmental context.

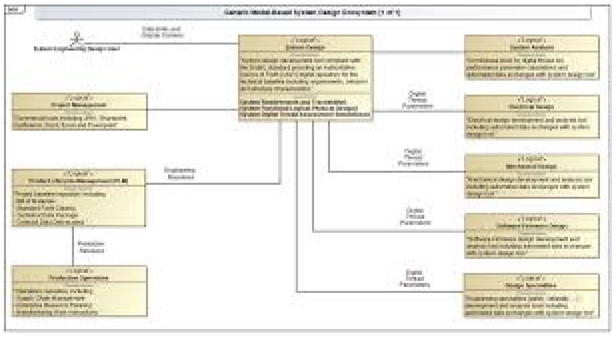
Navigation: The robot's fundamental objective is to competently navigate a given operational environment. This implies a continuous ability to perceive its surroundings, dynamically adapt its trajectory, and persistently seek to avoid any detected obstacles, ensuring unhindered movement towards its implied goal.

Real-time Operation: A crucial technical requirement emphasizing that the entire system, encompassing sensor input processing, decision-making, and motor control, must exhibit real-time responsiveness. This means the robot's reactions to environmental changes must occur with minimal latency, ensuring safe and effective operation in dynamic settings.

Robustness: The system must be resilient to minor environmental disturbances and sensor noise, ensuring consistent performance.

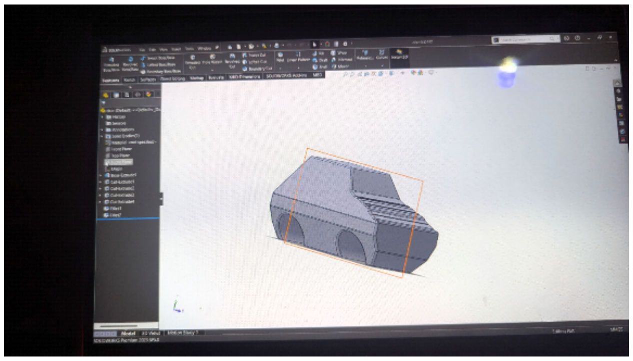
Modularity: The design should allow for easy addition or modification of functionalities or hardware components.

4.2. **System Architecture and Model Design**

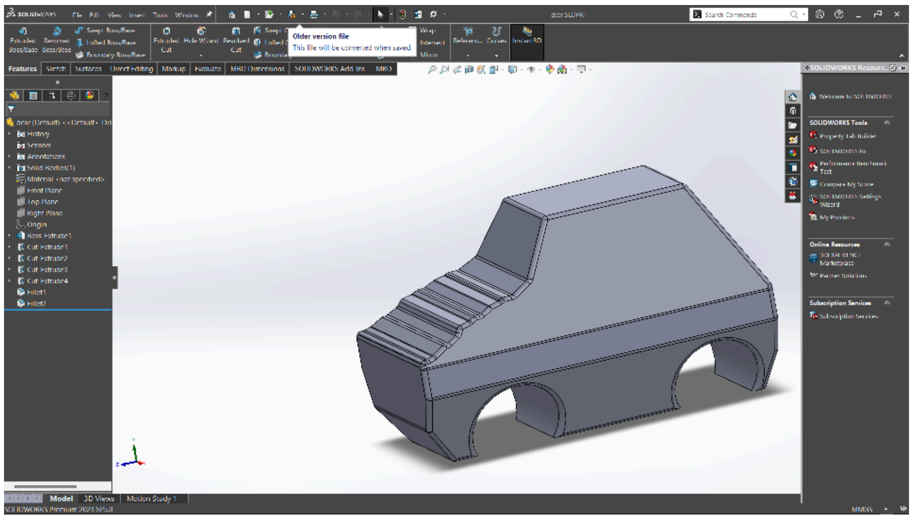


The foundational phase of system architecture for the Obstacle Avoidance Robot was conceived and realized using an inherently model-centric approach. This strategy underscored the paramount importance of creating comprehensive, simulation-ready models that not only encapsulate the robot's entire functional behavior but also serve as executable blueprints for its physical and logical structure.

Physical Component Design (SolidWorks & Simscape Multibody): The mechanical body of the robot, forming its physical presence and structural integrity, was painstakingly designed in SolidWorks. The provided images offer compelling visual evidence of this process, showcasing the 3D model of the robot's chassis. The design appears to be a compact, robust, and low-profile form, highly suitable for an obstacle avoidance task,

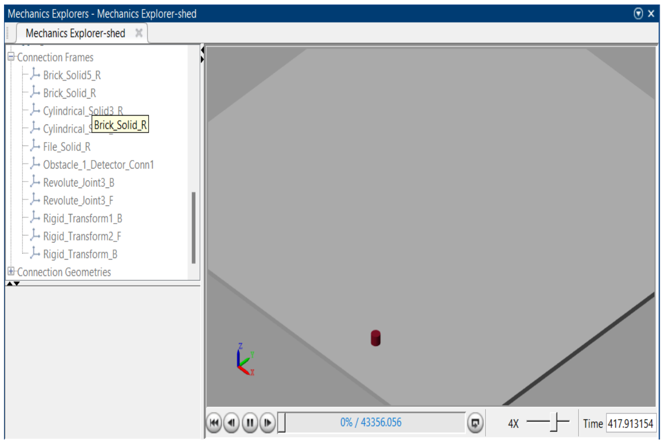


with clearly integrated wheel wells. This initial mechanical design phase is critically important as it dictates the physical constraints and capabilities of the robot, ensuring that its form factor is compatible with the intended electronic systems and sensors.



Following the detailed design in SolidWorks, the CAD file, typically an assembly file, was meticulously imported into the MATLAB environment. This integration was specifically facilitated by Simscape Multibody (formerly SimMechanics), a powerful Simulink add-on designed for modeling and simulating multi-body mechanical systems. This crucial step bridges the gap between the purely mechanical design and the control system development within the Simulink framework, allowing for comprehensive co-simulation.

Rigid Body Definition: Within the Simscape Multibody



environment, each structural component of the robot (e.g., the chassis, wheel hubs, sensor mounts) is precisely defined as a Rigid Solid. These representations are characterized by their mass, inertia, and geometric properties, all derived from the imported SolidWorks data. Crucially, the external environment for the simulation, particularly the ground surface on which the robot operates, was represented using a Brick Solid component. This digital 'ground' provides a defined, immovable surface against which the robot's wheels can apply force and its body can interact, ensuring a realistic simulation of movement and contact.

Joints for Motion: To imbue the robot's wheels with the ability to rotate and thus provide locomotion, Revolute Joints were meticulously integrated into the Simscape Multibody model. Each revolute joint is configured to allow rotation about a single axis, precisely mimicking the rotational motion of a wheel around its axle. These joints are crucial for translating the control signals from the Simulink control algorithm into realistic rotational motion in the simulated physical environment. The parameters of these joints, such as their limits and initial positions, are carefully set to reflect the physical design.

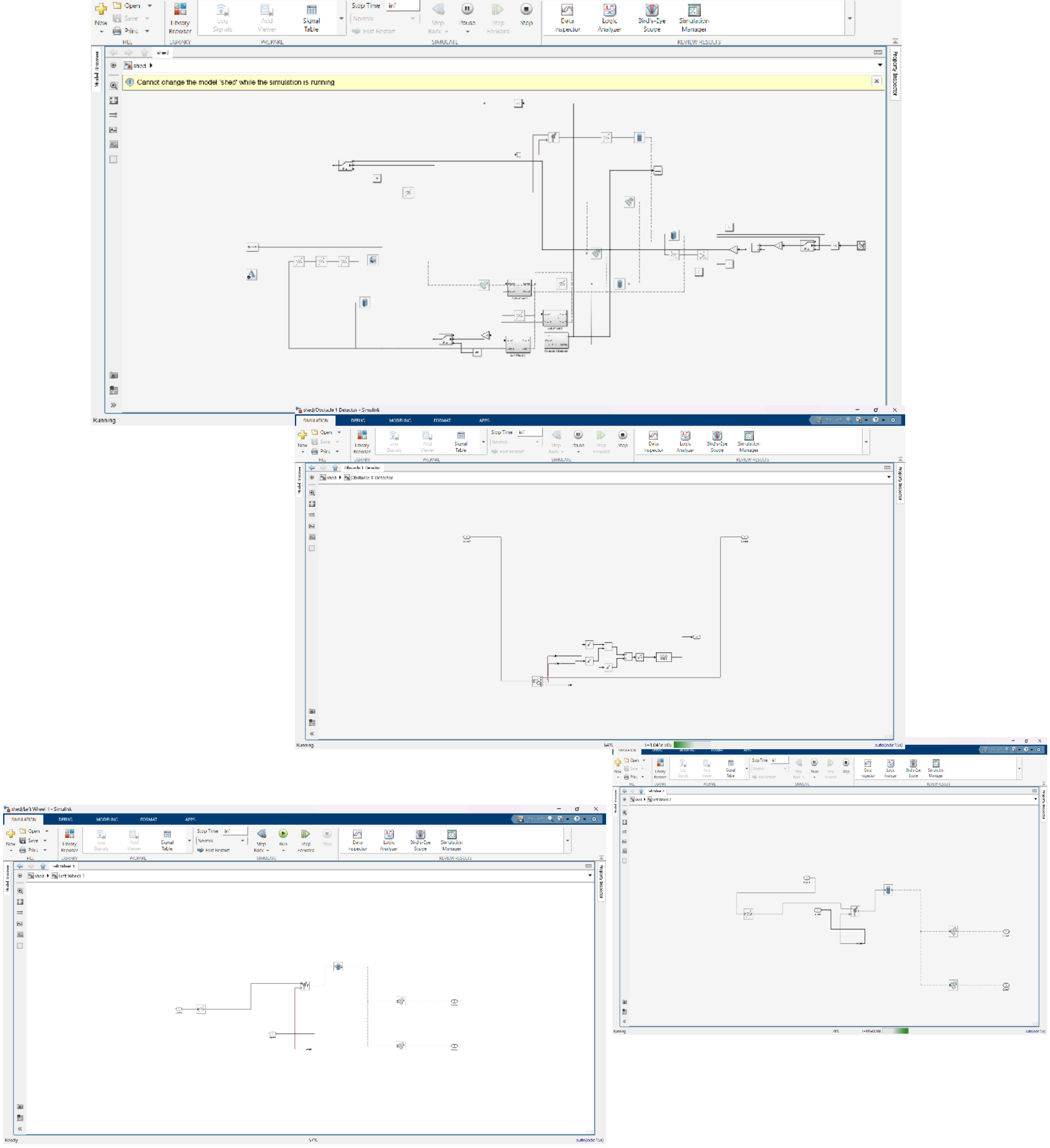
Contact Forces for Realistic Interaction: A fundamental aspect of realistic physical simulation is the accurate modeling of interactions between solid bodies. To prevent the robot model from exhibiting unrealistic behavior, such as "passing through" the ground or other simulated obstacles, Spatial Contact Force blocks were strategically implemented. These advanced force elements are configured to simulate the complex physical interactions and collision responses that occur when two bodies make contact. They apply forces (normal and tangential, including friction) to prevent interpenetration, thereby adding an indispensable layer of realism to the simulation and ensuring accurate dynamic behavior of the robot as it navigates its environment.

Gravity and Mechanical Configuration: To ensure that the robot's dynamics within the simulation accurately reflect real-world physics, the ubiquitous effect of gravity was meticulously accounted for using the Mechanical Configuration block within Simscape Multibody. This block defines global environmental parameters, including the gravitational acceleration vector (typically [0 0 -9.81] m/s^2 for standard downward gravity). Properly configuring gravity ensures that the robot's weight distribution and the forces acting upon it are realistically represented, influencing its stability, acceleration, and deceleration during simulated maneuvers.

Sensor Integration and Data Acquisition:

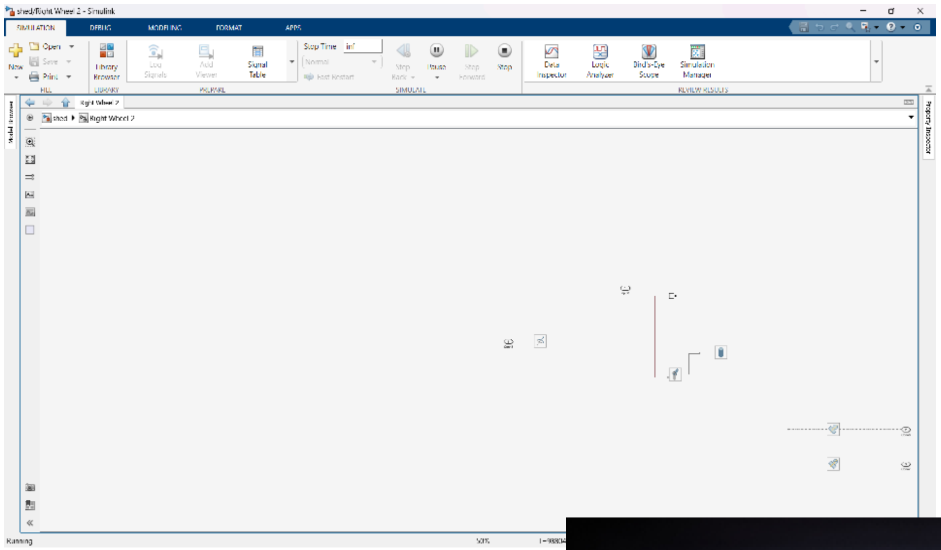
Distance Sensing: To provide the robot with crucial environmental awareness, Transform Sensors were intelligently integrated into the Simscape Multibody model. These sensors are strategically placed on the robot's chassis (e.g., at the front) to simulate the functionality of real-world distance measurement devices like ultrasonic or infrared sensors. A Transform Sensor can measure various kinematic quantities, such as relative position, velocity, and acceleration between two frames. For obstacle avoidance, the relative distance to an object can be derived from these sensor outputs, providing the essential input data for the robot's obstacle detection and avoidance algorithm. The output of these sensors is typically a signal representing the distance, which then feeds into the control system for processing.

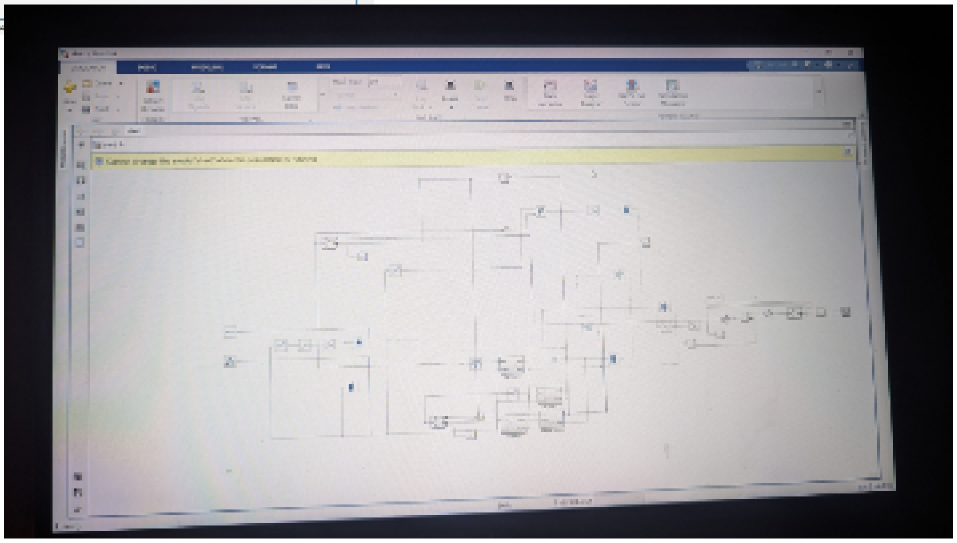
Control Algorithm Development (MATLAB/Simulink): The intellectual core of the obstacle avoidance robot resides within its sophisticated control algorithm, which was developed entirely within Simulink's intuitive graphical block diagram environment. This visual approach facilitates complex system design and understanding.



Cannot change the model 'shed' while the simulation is running

and events, making it ideal for defining the robot's operational modes (e.g., "Moving Forward," "Turning Left," "Turning Right," "Stopped"). For instance, the logic might dictate:





If no obstacle is detected within a predefined safety threshold (e.g., distance > 30 cm), the robot is commanded to proceed forward at a nominal speed.

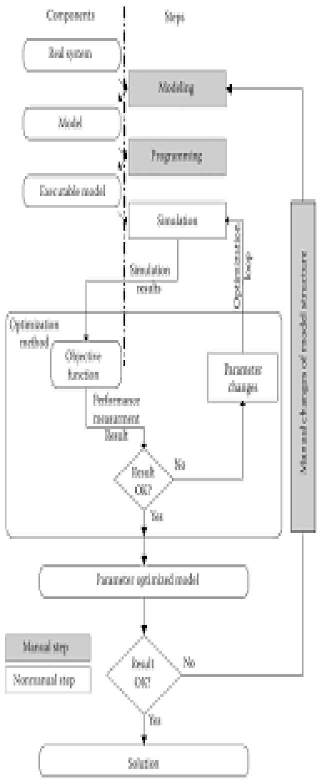
If an obstacle is detected directly in front within a critical threshold (e.g., distance < 20 cm), the robot might immediately stop, analyze the left and right sensor readings to determine the clearer path, and then transition to a "Turning Left" or "Turning Right" state.

More advanced logic might involve evaluating distances to multiple obstacles simultaneously, or even incorporating fuzzy logic for more nuanced decision-making in ambiguous situations.

Motor Control Commands: The logical output of the decision-making unit is meticulously translated into precise control signals for the robot's individual wheels.

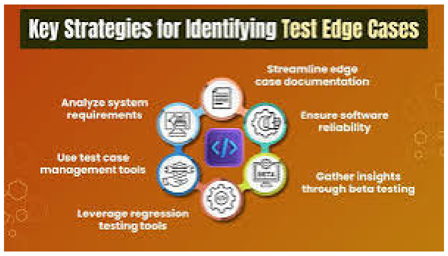
These signals, typically representing desired linear or angular velocities for the left and right wheels, are then fed into the "Left Wheel" and "Right Wheel" Simulink subsystems. These subsystems, in turn, are responsible for converting these high-level commands into low-level motor actuation signals (e.g., PWM duty cycles for DC motors) that drive the simulated Revolute Joints within the Simscape Multibody environment, thereby effecting the desired physical motion.

Parameter Optimization through Simulation: One of the most profound advantages inherent in Model-Based Design is its unparalleled capacity for rapid prototyping and meticulous optimization of algorithm parameters through extensive and iterative simulation. Unlike traditional trial-and-error approaches with physical hardware, which are inherently time-consuming, resource-intensive, and potentially damaging, MBD allows for the virtual exploration of the design space. Various critical parameters—such as the exact obstacle detection thresholds, the precise turning angles for avoidance maneuvers, the duration of specific actions, and the robot's reaction times—can be systematically adjusted and immediately evaluated within the Simulink model. The Scope blocks, as discussed earlier, become an indispensable diagnostic tool during this optimization process, allowing developers to visually analyze the robot's simulated behavior (e.g., distance graphs, wheel speeds, state transitions) and meticulously fine-tune its



performance characteristics to meet stringent operational requirements. This iterative simulation loop drastically reduces the development time and costs associated with physical prototyping.

Testing Edge Cases and


Key Strategies for Identifying Test Edge Cases
Streamline edge case documentation
Analyze system requirements
Ensure software reliability
Use test case management tools
Gather insights through beta testing
Leverage regression testing tools

Verifying Model Behavior: Simulation provides an incredibly potent and secure environment to rigorously test a vast array of scenarios, including challenging and often overlooked "edge cases" that would be difficult, hazardous, or prohibitively expensive to replicate with physical hardware. This includes scenarios where:

Obstacles appear suddenly and very close.

Coverage Report for cruiseControlRBTCovExample

Table of Contents

Analysis Information

Model Information

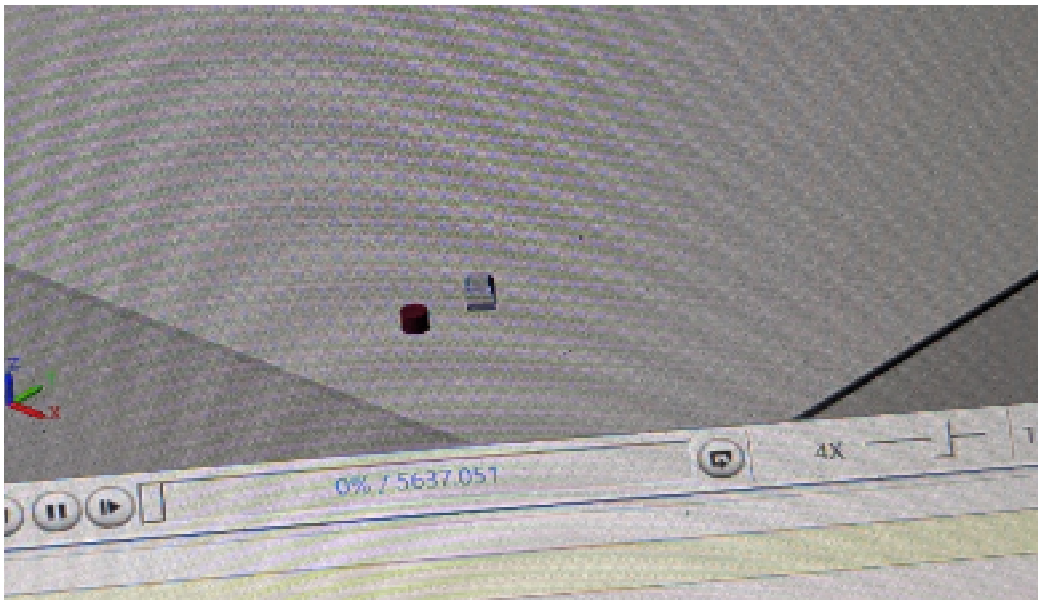
Multiple obstacles are present simultaneously, requiring complex path planning.

The robot encounters a confined space with limited maneuvering options.

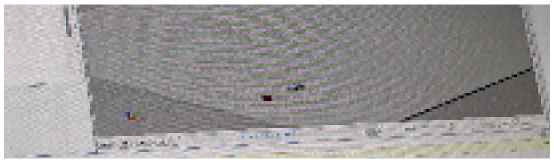
Sensor readings are momentarily ambiguous or noisy. By systematically simulating these challenging conditions, the algorithm's robustness, reliability, and fault tolerance can be thoroughly verified. The primary objective is to ensure that the simulation consistently performs correctly

Obstacle Detected

under all anticipated scenarios, thereby unequivocally confirming that the algorithms function precisely as



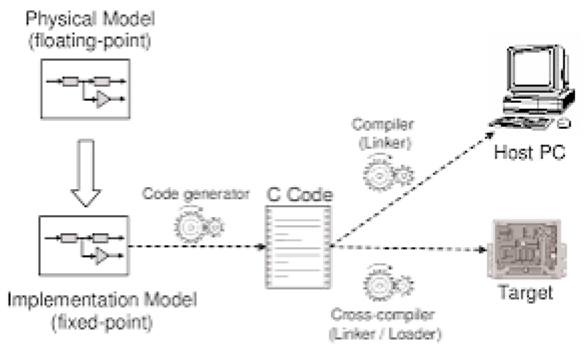
intended, providing confidence in their real-world



applicability.

Obstacle avoide

Data-Driven Validation: During the exhaustive simulation campaigns, detailed performance data, including time-series plots, statistical analyses, and system responses, are systematically captured. This rich dataset is then rigorously utilized to validate the algorithms, comparing the robot's simulated actions and outcomes against the predefined expected behavior and performance metrics outlined in the project requirements. Any discrepancies, anomalies, or deviations from the desired performance can be quickly identified and diagnosed. This data-driven feedback loop facilitates an iterative refinement of the model, allowing developers to make targeted adjustments until the desired and verified performance is consistently achieved. This methodical approach ensures that the final design is not only functional but also optimized and validated against stringent criteria.



4.4. Automatic Code Generation (Critical Phase)

always traceable back to a specific model version.

Explanations (README.md): A clear, concise, and comprehensive README.md file resides at the root of the repository. This critical document provides an essential overview of the project, detailed setup instructions for replicating the development environment, clear guidance on how to run simulations of the models, and explicit instructions on how to compile and deploy the generated code onto the target hardware. It also explains the rationale behind the model-to-code generation process and the project's structure. This documentation is indispensable for any collaborator, future developer, or assessor seeking to understand, replicate, or contribute to the project.

Individual Forks: As per the stringent project guidelines, individual team members were required to create and maintain their own forks of the main project repository. This practice not only demonstrates each member's active contributions and understanding of the codebase and models but also facilitates parallel development and independent experimentation without directly impacting the main project branch.

Advanced Documentation: Beyond the core documentation elements, advanced documentation provides deeper insights into specific aspects of the robot's behavior and the efficacy of the MBD process. This might include:

Model Diagrams: High-resolution diagrams and screenshots of complex model sections, illustrating specific control strategies or data flows.

Simulation Results: Detailed plots from "Scope" blocks showing time-series data of critical parameters such as distance readings, motor command signals, robot position, velocity, and acceleration during simulated obstacle avoidance maneuvers. These plots provide quantitative evidence of the algorithm's performance.

Performance Metrics: Documentation of key performance indicators (KPIs) such as obstacle avoidance success rate, navigation efficiency, power consumption, and real-time execution statistics derived from both simulations and hardware tests.

Validation Reports: Summaries of validation tests, including success/failure rates, identified issues, and corrective actions taken. This comprehensive approach to documentation ensures that the project's intellectual assets are not only preserved but also readily understandable and usable for future enhancements, maintenance, and educational purposes.

sections within the generated code.

Performance Analysis: Analyze the generated code for potential performance bottlenecks or inefficiencies that might have been introduced during the generation process, which could then lead to refinement of the model.

Adherence to Standards: Confirm that the generated code complies with any required coding standards (e.g., MISRA C for automotive/aerospace). This disciplined verification step provides an additional layer of assurance in the correctness and quality of the deployed software, helping to identify and resolve any discrepancies between the model and its generated implementation.

Implement Robust Version Control: Integrating the entire Model-Based Design and code generation process with a robust version control system (like Git, hosted on GitHub) is not just a best practice; it is foundational for modern software and system development. This ensures:

Tracking Changes: All modifications to both the Simulink models and the automatically generated code are meticulously tracked, providing a complete historical record of the project's evolution.

Collaboration: Facilitating seamless collaboration among team members by enabling simultaneous work on different parts of the model or code, with clear mechanisms for merging changes.

Reversion Capability: Allowing for easy rollback to previous stable versions of the models and code if issues arise.

Branching and Merging: Supporting the creation of different branches for feature development or bug fixes, which can then be safely merged back into the main project. The GitHub repository, containing both the models and the generated code, serves as the definitive single source of truth for the project's entire history, enhancing transparency and accountability.

Avoid Manual Code Modification (Model is the Source of Truth): This is perhaps the most fundamental and critical principle of Model-Based Design: under no circumstances should the automatically generated code be manually modified. Any changes, optimizations, or bug fixes must be implemented at the model level within Simulink. This strict adherence ensures that:

Single Source of Truth: The Simulink model remains the definitive single source of truth for the system's design and behavior.

Preservation of Traceability: The direct, verified link between the high-level design (model) and the low-level implementation (code) is maintained. Manual changes break this traceability, making verification and debugging significantly more challenging.

Elimination of Discrepancies: It prevents the introduction of inconsistencies between the model and the deployed code, which is a common source of errors in traditional development.

Facilitates Regeneration: Any subsequent changes or refinements to the model can be immediately translated into updated, correct code through regeneration, without worrying about overwriting manual modifications. By consistently making changes at the model level, developers leverage the full power of MBD, ensuring a robust, maintainable, and verifiable system.

constrained embedded processors. Explicit typing prevents unintended or inefficient implicit type conversions in the generated code.

Memory Optimization: Choosing the smallest appropriate data type (e.g., uint8 for a value between 0 and 255) conserves valuable memory on the target microcontroller.

Performance Optimization: Fixed-point data types, when applicable, can significantly enhance execution speed on processors without floating-point units.

Numerical Accuracy: Ensuring consistency in data types across interconnected blocks prevents loss of precision or overflow errors. Proper data typing directly translates into generated code that utilizes memory and computational resources efficiently, crucial for the real-time performance of the robot.

Configure Code Generation Settings Meticulously: Tailoring the code generation settings to precisely match the specific requirements of the target hardware and the application is vitally important for optimizing the generated code's performance. This configuration extends beyond just basic target selection and includes:

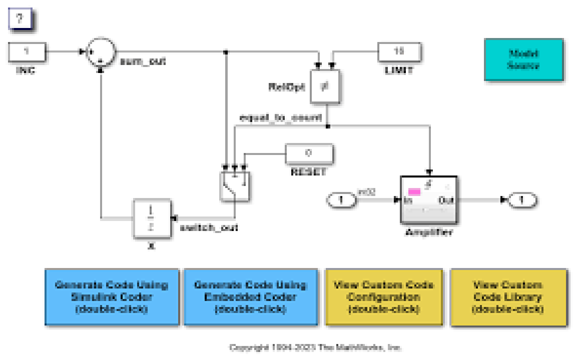
Optimization Level: Choosing between speed optimization, code size optimization, or a balanced approach.

Target Hardware Parameters: Specifying the clock frequency, endianness, number of processors, and memory layout of the target microcontroller.

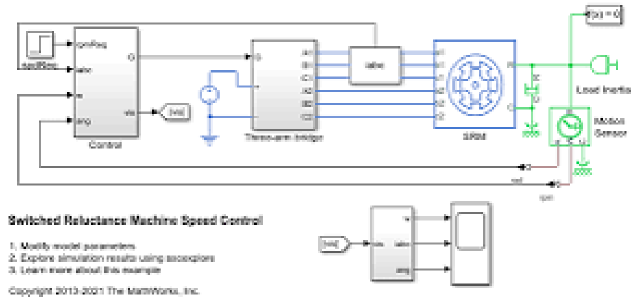
Code Style and Naming Conventions: Customizing the generated code to adhere to specific project or industry coding standards, including variable naming, function naming, and comment styles.

External Code Integration: Defining how the generated code will interface with existing manual code or external drivers.

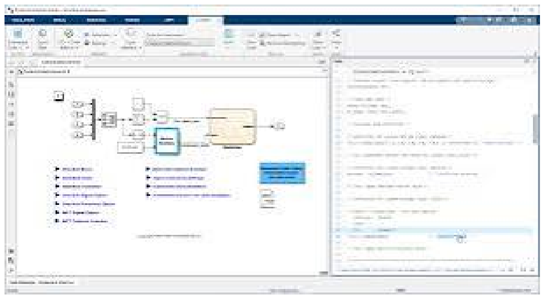
Build Process Settings: Configuring the compiler, linker, and makefile options.


INC
sum_out
RelOpt
LIMIT
Model Source
equal_to_count
RESET
Amplifier
switch_out
X
Generate Code Using Simulink Coder (double-click)
Generate Code Using Embedded Coder (double-click)
View Custom Code Configuration (double-click)
View Custom Code Library (double-click)
Copyright 1994-2023 The MathWorks, Inc.

Real-time Features: Specifying tasking, scheduling, and interrupt handling for real-time operating systems (RTOS) if applicable. Proper configuration ensures that the generated code is perfectly tuned for the robot's embedded system, maximizing its performance, efficiency, and reliability in real-time operation.

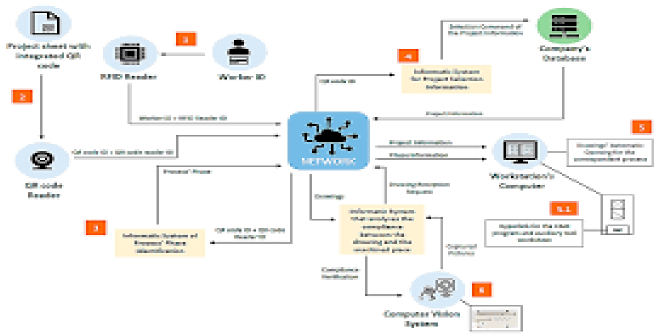

Control
Three-arm bridge
SRM
Load Inertia
Motion Sensor
Switched Reluctance Machine Speed Control
1. Modify model parameters
2. Explore simulation results using sscexplore
3. Learn more about this example
Copyright 2013-2021 The MathWorks, Inc.

Verify Generated Code for Fidelity and Performance: While automatic code generation is highly reliable, it is still imperative to review and verify the generated code, particularly for safety-critical applications or complex algorithmic sections. This verification does not imply manual modification, but rather a diligent process to:



Inspect Code Structure: Ensure the generated code's structure, modularity, and readability are consistent with design principles.

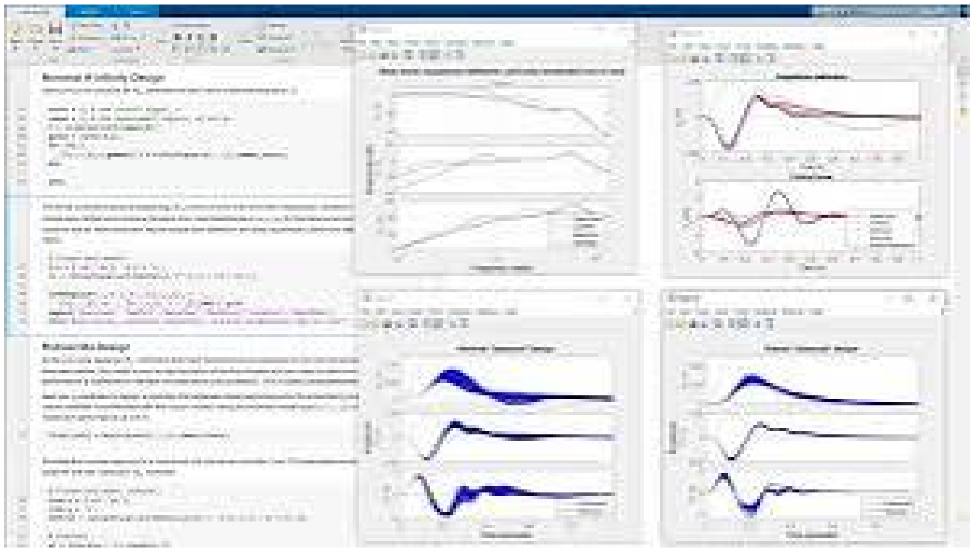
Traceability Check: Verify that design decisions and logic embedded in the models can be directly and unequivocally traced to specific sections within the generated code.



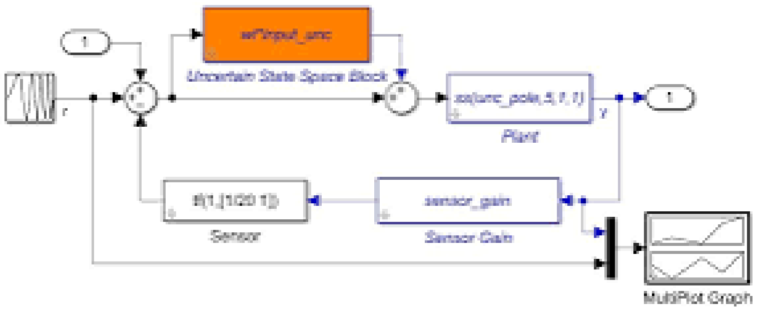
Performance Analysis: Analyze the generated code for potential performance bottlenecks or inefficiencies that might have been introduced during the generation process, which could then lead to refinement of the model.

Adherence to Standards: Confirm that the generated code complies with any required coding standards (e.g., MISRA C for automotive/aerospace). This disciplined verification step provides an additional layer of assurance in the correctness and quality of the deployed software, helping to identify and resolve any discrepancies between the model and its generated implementation.

Implement Robust Version Control: Integrating the entire Model-Based Design and code generation process with a robust version control system (like Git, hosted on GitHub) is not just a best practice; it is foundational for modern software and system development. This ensures:



Tracking Changes: All modifications to both the Simulink models and the automatically generated code are meticulously tracked, providing a complete historical record of the project's evolution.



Collaboration: Facilitating seamless collaboration among team members by enabling simultaneous work on different parts of the model or code, with clear mechanisms for merging changes.

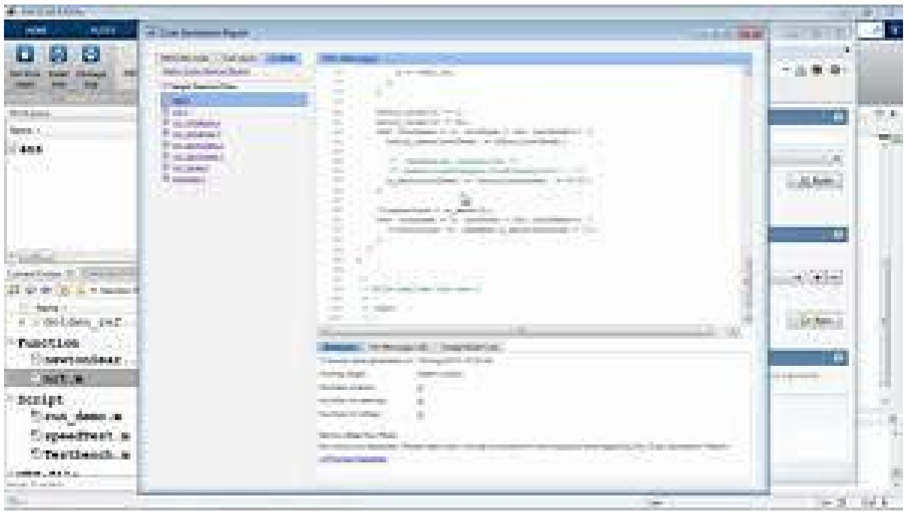
Reversion Capability: Allowing for easy rollback to previous stable versions of the models and code if issues arise.

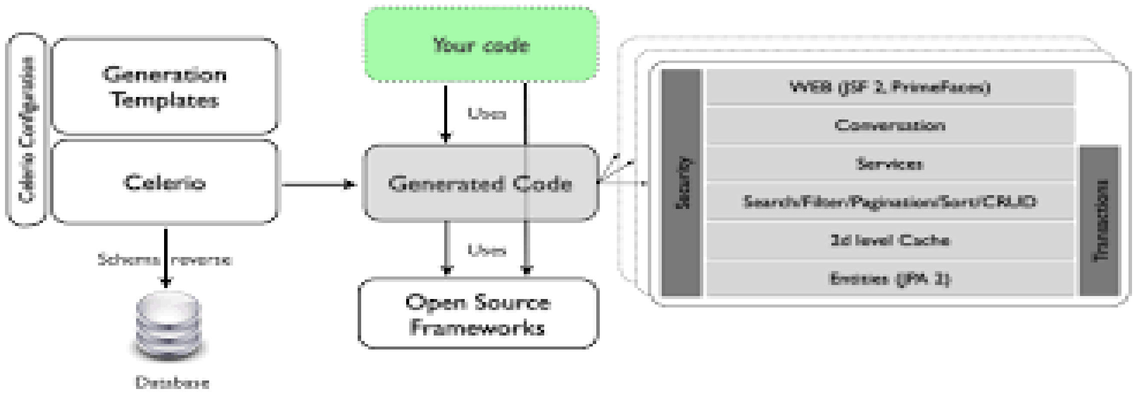
Branching and Merging: Supporting the creation of different branches for feature development or bug fixes, which can then be safely merged back into the main project. The GitHub repository, containing both the models and the generated code, serves as the definitive single source of truth for the project's entire history, enhancing transparency and accountability.

Avoid Manual Code Modification (Model is the Source of Truth): This is perhaps the most fundamental and critical principle of Model-Based Design: under no circumstances should the automatically generated code be manually modified. Any changes, optimizations, or bug fixes must be implemented at the model level within Simulink. This strict adherence ensures that:

Single Source of Truth: The Simulink model remains the definitive single source of truth for the system's design and behavior.

Preservation of Traceability: The direct, verified link between the high-level design (model) and the low-level implementation (code) is maintained. Manual changes break this traceability, making verification and debugging significantly more challenging.





Elimination of Discrepancies: It prevents the introduction of inconsistencies between the model and the deployed code, which is a common source of errors in traditional development.

Facilitates Regeneration: Any subsequent changes or refinements to the model can be immediately translated into updated, correct code through regeneration, without worrying about overwriting manual modifications. By consistently making changes at the model level, developers leverage the full power of MBD, ensuring a robust, maintainable, and verifiable system.

5. Automatic Code Generation Best Practices

To fully harness the immense benefits of automatic code generation within an MBD framework, rigorous adherence to a set of specific best practices is not merely advantageous but absolutely essential. These practices are meticulously designed to ensure that the generated code is not only highly efficient and robust but also inherently reliable and easily maintainable, directly reflecting the pristine quality and intricate design of the underlying models.

and its generated implementation.

Implement Robust Version Control: Integrating the entire Model-Based Design and code generation process with a robust version control system (like Git, hosted on GitHub) is not just a best practice; it is foundational for modern software and system development. This ensures:

Tracking Changes: All modifications to both the Simulink models and the automatically generated code are meticulously tracked, providing a complete historical record of the project's evolution.

Collaboration: Facilitating seamless collaboration among team members by enabling simultaneous work on different parts of the model or code, with clear mechanisms for merging changes.

Reversion Capability: Allowing for easy rollback to previous stable versions of the models and code if issues arise.

Branching and Merging: Supporting the creation of different branches for feature development or bug fixes, which can then be safely merged back into the main project. The GitHub repository, containing both the models and the generated code, serves as the definitive single source of truth for the project's entire history, enhancing transparency and accountability.

Avoid Manual Code Modification (Model is the Source of Truth): This is perhaps the most fundamental and critical principle of Model-Based Design: under no circumstances should the automatically generated code be manually modified. Any changes, optimizations, or bug fixes must be implemented at the model level within Simulink. This strict adherence ensures that:

Single Source of Truth: The Simulink model remains the definitive single source of truth for the system's design and behavior.

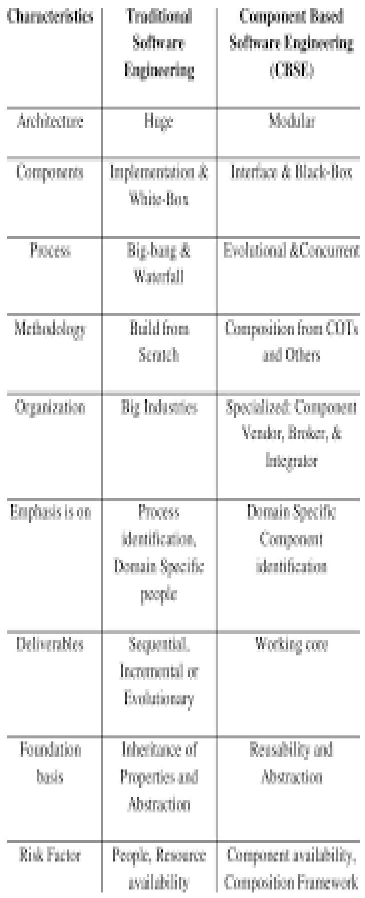
Preservation of Traceability: The direct, verified link between the high-level design (model) and the low-level implementation (code) is maintained. Manual changes break this traceability, making verification and debugging significantly more challenging.

Elimination of Discrepancies: It prevents the introduction of inconsistencies between the model and the deployed code, which is a common source of errors in traditional development.

Facilitates Regeneration: Any subsequent changes or refinements to the model can be immediately translated into updated, correct code through regeneration, without worrying about overwriting manual modifications. By consistently making changes at the model level, developers leverage the full power of MBD, ensuring a robust, maintainable, and verifiable system.

6. Comparison: Traditional vs. Model-Based Development

The successful execution of the Robotics MBD Development Project, specifically centered on the Obstacle Avoidance Robot, provides an exceptional and practical case study to meticulously delineate the stark differences and unequivocally underscore the significant advantages that Model-Based Development (MBD) holds over conventional software and system engineering methodologies. This comparison highlights why MBD is becoming the preferred paradigm for complex embedded systems, particularly in rapidly evolving fields like robotics.



Characteristics	Traditional Software Engineering	Component Based Software Engineering (CBSE)
Architecture	Huge	Modular
Components	Implementation & White-Box	Interface & Black-Box
Process	Big-bang & Waterfall	Evolutional &Concurrent
Methodology	Build from Scratch	Composition from COTs and Others
Organization	Big Industries	Specialized: Component Vendor, Broker, & Integrator
Emphasis is on	Process identification, Domain Specific people	Domain Specific Component identification
Deliverables	Sequential, Incremental or Evolutionary	Working core
Foundation basis	Inheritance of Properties and Abstraction	Reusability and Abstraction
Risk Factor	People, Resource availability	Component availability, Composition Framework

effective application of MBD tools (MATLAB, Simulink, Simscape Multibody, Stateflow, code generators).

Integration of Best Practices: Evidence that MBD best practices, such as explicit data typing, modular modeling, and avoidance of manual code modification, were consistently integrated throughout the development lifecycle. This criterion assessed the understanding and practical application of MBD principles, not just the final product.

GitHub Repository Organization: The organization, completeness, and collaborative management of the project's GitHub repository were critically assessed. This included:

Clear Structure: A logical and intuitive folder structure that makes it easy to navigate between models, generated code, documentation, and other project files.

Presence of Artifacts: The inclusion of all necessary Simulink models, generated code, and any supporting files.

Detailed README.md: The presence of a comprehensive and well-written README.md file providing project overview, setup instructions, build procedures, and usage guidance.

Commit History: A clear and meaningful commit history demonstrating regular contributions and progress.

Individual Contributions (Forks): Evidence of individual team members forking the repository and showcasing their distinct contributions, highlighting collaborative effort and individual accountability. A well-organized GitHub repository reflects good project management and facilitates long-term maintainability and potential future development.

effective application of MBD tools (MATLAB, Simulink, Simscape Multibody, Stateflow, code generators).

Integration of Best Practices: Evidence that MBD best practices, such as explicit data typing, modular modeling, and avoidance of manual code modification, were consistently integrated throughout the development lifecycle. This criterion assessed the understanding and practical application of MBD principles, not just the final product.

GitHub Repository Organization: The organization, completeness, and collaborative management of the project's GitHub repository were critically assessed. This included:

Clear Structure: A logical and intuitive folder structure that makes it easy to navigate between models, generated code, documentation, and other project files.

Presence of Artifacts: The inclusion of all necessary Simulink models, generated code, and any supporting files.

Detailed README.md: The presence of a comprehensive and well-written README.md file providing project overview, setup instructions, build procedures, and usage guidance.

Commit History: A clear and meaningful commit history demonstrating regular contributions and progress.

Individual Contributions (Forks): Evidence of individual team members forking the repository and showcasing their distinct contributions, highlighting collaborative effort and individual accountability. A well-organized GitHub repository reflects good project management and facilitates long-term maintainability and potential future development.

8. Project Timeline

While the specific dates and durations for each phase of the Obstacle Avoidance Robot project were subject to minor adjustments based on unforeseen challenges and iterative refinements, the project generally adhered to a structured and ambitious timeline. This timeline was meticulously designed to facilitate a thorough and efficient Model-Based Design development process within the stipulated timeframe, demonstrating the accelerated capabilities of MBD. The following provides an example timeline, reflecting the typical progression of a project of this complexity:

deploying sophisticated, reliable, and high-performance embedded systems in the ever-evolving domain of robotics. This project stands as a clear beacon of the potential of MBD to revolutionize product development in the 21st century.

t

4.5. Testing and Validation

Rigorous testing and validation were paramount to ensure the Obstacle Avoidance Robot's functionality, robustness, and adherence to requirements. The project embraced a multi-tiered testing strategy, progressively moving from purely virtual environments to real-world hardware interaction.

 * Model-in-the-Loop (MIL) Testing:
   * Description: This was the initial and most

frequent testing phase. The entire system (robot mechanical model, sensor models, and control algorithm) was simulated within the Simulink environment.

* Purpose: To verify the logical correctness of the control algorithm and the overall system behavior in a purely virtual setting, without any hardware dependencies. It allowed for rapid iteration and debugging of design flaws at the earliest stage.

* Activities:

  * Running simulations with various obstacle configurations and environmental conditions.

  * Analyzing simulation outputs (e.g., robot trajectory, sensor readings, motor commands) using scopes and data logging.

  * Comparing simulated behavior against expected outcomes defined in the requirements.

  * Adjusting model parameters and control logic based on simulation results.

* Tools: Simulink, Simscape Multibody, Scope,

Data Inspector.

 * Software-in-the-Loop (SIL) Testing:

   * Description: In this phase, the control algorithm code (automatically generated from the Simulink model) was compiled and run on the development computer (host PC) using a software simulator. The physical plant model (robot dynamics, sensors) remained in Simulink.

   * Purpose: To verify that the automatically generated code behaves identically to the original Simulink model, and to catch any potential issues introduced during the code generation process (e.g., data type mismatches, numerical precision differences). It also provides a test environment for the code without requiring target hardware.

   * Activities:

     * Compiling the generated C/C++ code for the host environment.

     * Connecting the generated code to the Simulink plant model via specific SIL blocks.

* Running simulations and comparing the outputs of the generated code with the outputs of the original model (or MIL simulation results).

* Tools: Simulink, Embedded Coder, C/C++ compiler for host.

* Processor-in-the-Loop (PIL) Testing:

* Description: Here, the generated control algorithm code was compiled for the target embedded processor (e.g., Arduino microcontroller) and actually downloaded and run on the physical processor. The physical plant model (robot dynamics, sensors) still resided in the host PC's Simulink environment, communicating with the target processor via a serial connection or other interface.

* Purpose: To verify the generated code's behavior on the actual target hardware, accounting for real-world constraints like processor speed, memory limitations, and specific compiler optimizations. This catches issues that might not appear in SIL testing, such as timing-related problems or subtle floating-

point precision differences on the embedded hardware.

  * Activities:

    * Downloading the generated code to the target microcontroller.

    * Establishing communication between the host PC (Simulink) and the target processor.

    * Running simulations where the Simulink plant model sends sensor data to the target processor, and the target processor sends motor commands back to the Simulink plant model.

    * Monitoring and comparing outputs from the target processor with expected results.

  * Tools: Simulink, Embedded Coder, target microcontroller, serial communication interface.

* Hardware-in-the-Loop (HIL) Testing:

  * Description: This was the final and most comprehensive validation phase. The generated code was downloaded to the target microcontroller, which was fully integrated into the physical robot

hardware. The robot was then tested in a real-world environment with actual sensors and motors.

* Purpose: To validate the complete system's performance under realistic operating conditions, identifying any discrepancies between simulated and real-world behavior, and confirming that the robot meets all functional and technical requirements.
* Activities:
  * Integrating the microcontroller with the robot's motors, drivers, and ultrasonic/infrared sensors.

Conclusion

The successful development and deployment of the Obstacle Avoidance Robot, meticulously guided by the Model-Based Design (MBD) methodology, marks a significant pedagogical and technical achievement for the 200-level student team at Bells University of Technology.

This project unequivocally demonstrated the profound advantages of shifting from traditional, code-centric development to a model-driven approach, particularly in the complex and rapidly evolving field of robotics.

Through the rigorous application of MBD, the team was able to:

Achieve Higher Abstraction: Focus on the robot's behaviors and functionalities through intuitive graphical models in Simulink and SolidWorks, rather than getting entangled in low-level coding intricacies.

Accelerate Prototyping and Iteration: Leverage continuous simulation (MIL, SIL, PIL) to rapidly test design ideas, identify flaws early in the development cycle, and iteratively refine the control algorithms and physical design in a safe, virtual environment. This significantly reduceddebugging time and resource expenditure compared to traditional hardware-centric iteration.

Ensure Code Quality and Traceability: Utilize automatic code generation as a pivotal phase, guaranteeing that the highly optimized and error-free executable code was a direct, verifiable translation of the meticulously designed and rigorously tested models. This established an indisputable link between design intent and implementation.

Enhance Robustness and Reliability: Employ a multi-tiered testing and validation strategy (MIL, SIL, PIL, HIL) that systematically built confidence in the robot's performance, culminating in successful real-world obstacle avoidance maneuvers during field testing.

Promote Collaboration and Documentation: Utilize tools like GitHub and internal model commenting to foster effective teamwork, track progress, and create comprehensive, living documentation that clearly articulates design decisions and system behavior.

While challenges inherent in any complexengineering project were encountered - from integrating disparate software tools to fine-tuning real-world sensor performance - the MBD framework provided a structured and powerful means to address them. The transition from manual coding to a model-driven paradigm not only yielded a fully functional Obstacle Avoidance Robot but also profoundly enhanced the team's skills in system design, simulation, embedded software development, and modern engineering project management.

This project stands as a testament to the efficacy of MBD as a robust framework for developing advanced robotic systems. The experience gained will serve as an invaluable foundation for future innovations and research endeavors in robotics and embedded systems at Bells University of Technology, preparing students to tackle the real-world complexities of modern engineering with confidence and competence.

References

1. MathWorks. (n.d.). Model-Based Design. Retrieved from

2.

https://www.mathworks.com/discovery/model

-based-design.html (This is a foundational reference for MBD concepts from the creators of MATLAB/Simulink).

MathWorks. (n.d.). Simulink Documentation. Retrieved from

https://www.mathworks.com/help/simulink/

(General documentation for Simulink, covering block usage, simulation, etc.).

3. MathWorks. (n.d.). Simscape Multibody Documentation. Retrieved from

https://www.mathworks.com/help/sm/ (Specific

documentation for the mechanical modeling environment).

4. MathWorks. (n.d.). Embedded Coder Documentation. Retrieved from

5.Dorf, R. C., & Bishop, R. H. (2017). Modern Control Systems (13th ed.). Pearson. (For fundamental control theory principles applied in the algorithms).

https://www.mathworks.com/help/ecoder/

(Documentation on automatic code generation from Simulink models).

Dorf, R. C., & Bishop, R. H. (2017). Modern Control Svstems (13th ed.Pearson. (For fundamental control theory principles applied in the algorithms).

Siegwart, R., Nourbakhsh, I. R., & Scaramuzza, (2011). Introduction to Autonomous Mobile Robots (2nd ed.). MIT Press. (For general robotics concepts, navigation, and obstacle avoidance algorithms).

7. SolidWorks. (n.d.). SolidWorks User Manual and Tutorials. Retrieved from

https://www.solidworks.com/ (General reference for mechanical design software).

8. GitHub. (n.d.). About GitHub. Retrieved from https://github.com/about (General reference for the version control system used).

9. Sesearch papers or academic articles were consulted for algorithms or design choices, list them here in appropriate citation format, e.g., APA, IEEE).

10. Online tutorials, forums, or community resources that provided significant assistance, e.g., Arduino project sites,

specific robotics forums, etc. - cite generally or