Table of Verified Properties

Property	Formula and Function	FSM→C Utility
D.1 Structural Safety Properties		
Property P1	**Formula** A[] not deadlock **Function** Verifies the absence of global deadlock in the system composed of master and slave automata. This property ensures that no system configuration leads to a blocked state in which no transition can be executed, thus guaranteeing continuous and effective protocol progression.	**FSM→C Utility** Essential for C implementation: ensures that no state combination exists where the code could block indefinitely. Guides event loop design and guarantees that at least one outgoing transition is always defined in each state. Allows identification of states requiring a timeout or watchdog to avoid blocking when expected messages are not received.
Property P2	**Formula** A[] ((MASTER_SDLC.slave_state_in_m == NRM and MASTER_BITBUS.irrec_err == 0) imply not deadlock) **Function** Verifies the absence of deadlock specifically in nominal mode (NRM) when the SDLC connection is established and there is no irrecoverable error. This property ensures that normal protocol operation never leads to blocking.	**FSM→C Utility** Crucial for nominal code: confirms that main execution paths (without SDLC errors) are always non-blocking. Allows optimization of production code by clearly separating nominal logic (guaranteed deadlock-free) from error handling. Justifies the absence of certain timeouts in nominal states, reducing code complexity.
D.2 Data Integrity and Protocol Validity		
Property P3	**Formula** MASTER_BITBUS.RECEIVE_ALPHANUM_S and cs_frame_s == cs_ref_s --> MASTER_BITBUS.PROCESS_BITBUS **Function** Verifies that when the master receives an alphanumeric frame from the slave with a valid checksum (cs_frame_s == cs_ref_s), it systematically transitions to the PROCESS_BITBUS processing state. Guarantees correct processing of valid frames.	**FSM→C Utility** Dictates checksum validation implementation: the C checksum verification function must be called immediately after reception, and if valid, trigger an unconditional transition to the PROCESS_BITBUS state. No additional intermediate verification is necessary. Structures the code as: if (checksum_valid(frame)) { state = PROCESS_BITBUS; }.

Property P4	**Formula** E<> MASTER_BITBUS.RECEIVE_ALPHANUM_S and cs_frame_s == cs_ref_s **Function** Verifies the reachability of the state where the master receives a valid alphanumeric frame from the slave. Confirms that at least one execution path leads to reception of a valid frame, demonstrating that the nominal scenario is possible.	**FSM→C Utility** Validates code testability: guarantees that a unit test scenario can be created where a valid frame is received. Serves as a basis for integration tests and nominal test cases. Indicates that this execution path must be covered by code coverage tests.
Property P5	**Formula** (MASTER_BITBUS.RECEIVE_ALPHANUM_S and cs_frame_s != cs_ref_s) --> MASTER_BITBUS.SDLC_CONNECTION_REQUEST **Function** Verifies that when the master receives an alphanumeric frame with an invalid checksum (cs_frame_s != cs_ref_s), it systematically transitions to the SDLC_CONNECTION_REQUEST state. Guarantees detection and appropriate handling of corrupted frames on the master side.	**FSM→C Utility** Determines checksum error handling: in case of invalid checksum, the C implementation must immediately perform a reset to SDLC_CONNECTION_REQUEST without attempting local recovery. Simplifies the code by avoiding retry mechanisms at this level. Structure: if (!checksum_valid(frame)) { state = SDLC_CONNECTION_REQUEST};
Property P6	**Formula** E<> (MASTER_BITBUS.RECEIVE_ALPHANUM_S and cs_frame_s != cs_ref_s) **Function** Verifies the reachability of the state where the master receives an alphanumeric frame with an invalid checksum. Confirms that the transmission error scenario is represented in the model.	**FSM→C Utility** Guides robustness test implementation: indicates that tests with injection of corrupted frames must be created. Confirms that error paths are reachable and must be tested. Justifies inclusion of data corruption tests in the validation suite.
Property P7	**Formula** (SLAVE_BITBUS.RECEIVE_DATA_FRAME and cs_frame_m == cs_ref_m and SLAVE_BITBUS.y == T_rep) --> SLAVE_BITBUS.SEND_ALPHANUM_S **Function** Verifies that when the slave receives a valid data frame from the master (correct checksum) and the response delay T_rep is reached, it systematically transitions to the alphanumeric response sending state. Guarantees compliance with response timing for valid frames.	**FSM→C Utility** Imposes strict temporal synchronization: the C implementation must use a timer (y) that reaches exactly T_rep before sending the response. Structure: if (checksum_valid && timer_y >= T_rep) { state = SEND_ALPHANUM_S; }. Avoids premature responses and guarantees deterministic timing conforming to the industrial protocol.
Property P8	**Formula** E<> (SLAVE_BITBUS.RECEIVE_DATA_FRAME and cs_frame_m == cs_ref_m and SLAVE_BITBUS.y == T_rep) **Function**	**FSM→C Utility** Validates nominal slave path testability: confirms that correct timing can be simulated. Guides implementation of temporal tests

	Verifies the reachability of the state where the slave receives a valid data frame and reaches the response delay. Confirms that the nominal slave→master communication scenario is possible.	with a timer for T_rep. Indicates that this scenario must be covered in temporal integration tests.
Property P9	**Formula** (SLAVE_BITBUS.RECEIVE_DATA_FRAME and cs_frame_m != cs_ref_m and SLAVE_BITBUS.y == T_rep) --> SLAVE_BITBUS.ERROR_IN_S **Function** Verifies that when the slave receives a data frame with an invalid checksum and reaches the response delay, it systematically transitions to the ERROR_IN_S error state. Guarantees detection of transmission errors on the slave side.	**FSM→C Utility** Specifies error behavior with timing: even if the checksum is invalid, the slave waits T_rep before signaling the error. Avoids overly rapid transitions that could disrupt the protocol. Code: if (!checksum_valid && timer_y >= T_rep) { state = ERROR_IN_S; log_error(); }.
Property P10	**Formula** E<> (SLAVE_BITBUS.RECEIVE_DATA_FRAME and cs_frame_m != cs_ref_m and SLAVE_BITBUS.y == T_rep) **Function** Verifies the reachability of the state where the slave receives an invalid data frame after the response delay. Confirms that the error scenario is modeled on the slave side.	**FSM→C Utility** Guides timed error tests: confirms that behavior can be tested when a corrupted frame arrives after waiting for the delay. Justifies tests combining data corruption and compliance with temporal constraints.
Property P11	**Formula** (SLAVE_BITBUS.RECEIVE_INVALID_FRM and cs_frame_m == cs_ref_m) --> SLAVE_BITBUS.SEND_DATA_RESPONSE and invalid_data_m == 1) **Function** Verifies that when the slave receives an invalid frame (incorrect format) but with a valid checksum, it still sends a data response, keeping the information about the invalidity of the previous frame. Guarantees signaling of format errors.	**FSM→C Utility** Distinguishes two validation levels: C code must implement two sequential validations. If format invalid but checksum OK, the slave responds with preservation of the invalid state of the frame received by the master silently: if (checksum_valid && invalid_data_m) { response.invalid_data = 1; state = SEND_DATA_RESPONSE; }.
Property P12	**Formula** E<> (SLAVE_BITBUS.RECEIVE_INVALID_FRM and cs_frame_m == cs_ref_m) **Function** Verifies the reachability of the state where the slave receives a frame with an invalid format but a correct checksum. Confirms that this protocol error scenario is represented.	**FSM→C Utility** Guides protocol validation tests: indicates that testing with syntactically invalid frames but with correct checksum is necessary. Validates coverage of high-level protocol errors.
Property P13	**Formula** (SLAVE_BITBUS.RECEIVE_INVALID_FRM and cs_frame_m != cs_ref_m) --> SLAVE_BITBUS.ERROR_IN_S	**FSM→C Utility** Specifies multiple error handling: in presence of combined errors, direct transition to ERROR_IN_S.

	Function Verifies that when the slave receives a frame that is both invalid in format AND with an incorrect checksum, it systematically transitions to the error state. Guarantees strict handling of multiple errors.	Simplifies code by avoiding separate handling of multiple errors. Code: if (!checksum_valid || !format_valid) { state = ERROR_IN_S; } with checksum priority.
Property P14	**Formula** E<> (SLAVE_BITBUS.RECEIVE_INVALID_FRM and cs_frame_m != cs_ref_m) **Function** Verifies the reachability of the state where the slave receives a frame with multiple errors (format and checksum). Confirms complete representation of error scenarios.	**FSM→C Utility** Justifies worst-case tests: validates that testing can be done with frames presenting several types of simultaneous errors. Guides implementation of extreme robustness tests.
Property P15	**Formula** (MASTER_BITBUS.RECEIVE_VALID_FRM and cs_frame_s == cs_ref_s) --> MASTER_BITBUS.PROCESS_BITBUS **Function** Verifies that when the master receives a valid frame (correct format) with a correct checksum, it systematically transitions to the processing state. Guarantees appropriate processing of protocol-compliant frames.	**FSM→C Utility** Confirms complete validation logic on master side: checksum AND format must be validated before processing. Structures reception code with two-step validation. Code: if (checksum_valid(frame) && format_valid(frame)) { state = PROCESS_BITBUS; } else error handling.
Property P16	**Formula** E<> (MASTER_BITBUS.RECEIVE_DATA_FRAME and cs_frame_s == cs_ref_s) **Function** Verifies the reachability of the state where the master receives a valid data frame with correct checksum. Confirms that slave→master communication can be performed correctly.	**FSM→C Utility** Validates code bidirectionality: confirms that the master reception path is functional. Guides implementation of bidirectional communication tests.
D.3 Temporal Constraints		
Property P17	**Formula** A[] (MASTER_BITBUS.SEND_PROCESS imply MASTER_BITBUS.x == T_Pol) **Function** Verifies that each time the master reaches the SEND_PROCESS state, its clock x is exactly equal to T_Pol (polling period). Guarantees strict compliance with cyclic polling timing defined by the industrial protocol.	**FSM→C Utility** Imposes strict periodic timing: C code must implement a precise cyclic timer (x). Sending can only occur at x == T_Pol exactly. Structure: if(timer_x >= T_Pol); state = SEND_PROCESS;;. Justifies use of high-precision timers to guarantee periodicity.
Property P18	**Formula** (MASTER_BITBUS.WAIT_BITBUS_RESP and MASTER_BITBUS.x == MASTER_BITBUS.T_out_M)	**FSM→C Utility** Specifies master timeout handling: at T_out_M expiration, two possible transitions depending on whether a

	--> (MASTER_BITBUS.PROCESS_BITBUS or MASTER_BITBUS.SDLC_CONNECTION_REQUEST) **Function** Verifies that when the master waits for a Bitbus response and its timeout T_out_M expires, it transitions either to processing (if response received in time), or to a new SDLC connection request. Guarantees that no timeout leaves the system in an indefinite waiting state.	response has been received. Code: if (timer_x >= T_out_M) { if (response_received) state = PROCESS_BITBUS; else state = SDLC_CONNECTION_REQUEST; }. Avoids infinite waits and guarantees deterministic recovery.
Property P19	**Formula** E<> (MASTER_BITBUS.WAIT_BITBUS_RESP and MASTER_BITBUS.x == MASTER_BITBUS.T_out_M) **Function** Verifies the reachability of the state where the master reaches its timeout while waiting for a response. Confirms that the master timeout scenario is modeled and reachable.	**FSM→C Utility** Guides timeout tests: validates that a timeout can be simulated by not providing a response within the T_out_M delay. Justifies non-response and slave disconnection tests.
Property P20	**Formula** (SLAVE_BITBUS.y >= SLAVE_BITBUS.T_out_S) --> SLAVE_BITBUS.T_OUT_EXCEEDED_S **Function** Verifies that when the slave's clock y reaches or exceeds timeout T_out_S, the slave systematically transitions to the T_OUT_EXCEEDED_S state. Guarantees detection and handling of timeout on the slave side.	**FSM→C Utility** Defines slave timeout: unconditional transition to T_OUT_EXCEEDED_S when y >= T_out_S. Code: if (timer_y >= T_out_S) { state = T_OUT_EXCEEDED_S; }. No need for complex recovery logic at this level, just detection and transition.
Property P21	**Formula** E<> (SLAVE_BITBUS.y >= SLAVE_BITBUS.T_out_S) **Function** Verifies the reachability of the state where the slave reaches its timeout. Confirms that the slave timeout scenario is represented in the model.	**FSM→C Utility** Guides slave timeout tests: validates that an absence of master communication can be simulated.
Property P22	**Formula** A[] (SLAVE_BITBUS.SEND_LINK_RESP imply SLAVE_BITBUS.y == T_rep) **Function** Verifies that each time the slave sends a link response (LINK_RESP), its clock y is exactly equal to the response delay T_rep. Guarantees compliance with response timing for link requests.	**FSM→C Utility** Imposes fixed response delay for LINK_RESP: the slave must wait exactly T_rep before sending. Code: wait_until(timer_y == T_rep); send_link_response(); timer_y = 0;. Guarantees predictable temporal behavior conforming to industrial specifications.
Property P23	**Formula** A[] (SLAVE_BITBUS.SEND_UNLINK_RESP imply SLAVE_BITBUS.y == T_rep) **Function** Verifies that each time the slave sends an unlink response (UNLINK_RESP), its clock y is exactly equal	**FSM→C Utility** Unifies timing of all responses: UNLINK_RESP uses the same delay T_rep as LINK_RESP. Simplifies implementation by using a generic response delay function

	to T_rep. Guarantees compliance with response timing for unlink requests.	for all message types. Reusable code: response_delay(T_rep); send_unlink_response();
Property P24	**Formula** A[] (SLAVE_BITBUS.SEND_DATA_RESPONSE imply SLAVE_BITBUS.y == T_rep) **Function** Verifies that each time the slave sends an alphanumeric response, its clock y is exactly equal to T_rep. Guarantees compliance with response timing for all data requests.	**FSM→C Utility** Confirms temporal uniformity: all slave responses (LINK, UNLINK, DATA) use T_rep. Allows implementation of a single timing management function for all responses, reducing code complexity and error risks.
D.4 Liveness Properties		
Property P25	**Formula** MASTER_BITBUS.PROCESS_BITBUS --> MASTER_BITBUS.SEND_PROCESS **Function** Verifies that each time the master reaches the PROCESS_BITBUS processing state, it will eventually reach the SEND_PROCESS state. Guarantees absence of livelock and ensures communication cycle progression without permanent blocking in the processing state.	**FSM→C Utility** Guarantees code progression: the PROCESS_BITBUS state cannot be a terminal state. C implementation must ensure that after processing, there is always a transition to SEND_PROCESS. Avoids infinite loops in processing. Structure: process_data(); state = SEND_PROCESS; with prohibition of remaining indefinitely in PROCESS_BITBUS.
Property P26	**Formula** MASTER_BITBUS.WAIT_FOR_CONNECTION and SLAVE_SDLC.ACK --> MASTER_BITBUS.PROCESS_BITBUS **Function** Verifies that when the master waits for a connection and the SDLC slave sends an acknowledgment (ACK), the master will eventually reach the PROCESS_BITBUS state. Guarantees correct resumption of the Bitbus protocol after SDLC connection establishment.	**FSM→C Utility** Specifies inter-layer synchronization: after receiving an SDLC ACK, transition to PROCESS_BITBUS to resume Bitbus. Code: if (state == WAIT_FOR_CONNECTION && sdlc_ack_received()) { state = PROCESS_BITBUS; }. Ensures coherence between SDLC and Bitbus layers.
D.5 Resource Bounds		
Property P27	**Formula** A[] (MASTER_BITBUS.attempt_bb <= 2) **Function** Verifies that the Bitbus attempt counter (attempt_bb) never exceeds 2. Guarantees an upper bound on the number of retransmissions to avoid infinite retry loops and ensure deterministic resource usage.	**FSM→C Utility** Bounds Bitbus retries: C code must implement an attempt_bb counter with strict verification <= 2. Code: if (++attempt_bb > 2) { abandon_and_reset(); } else { retry_transmission(); }. Guarantees deterministic termination and avoids resource exhaustion (CPU,

		bandwidth) in case of persistent failure.
Property P28	**Formula** A[] (MASTER_BITBUS.attempt_link <= 3) **Function** Verifies that the link attempt counter (attempt_link) never exceeds 3. Guarantees an upper bound on the number of connection attempts to avoid infinite retries in case of persistent slave failure.	**FSM→C Utility** Bounds link attempts: maximum 3 LINK attempts. Code: if (++attempt_link > 3) { report_link_failure(); state = ERROR_STATE; } else { resend_link_request(); }. Allows rapid detection of slave failure and alerting of supervisor system.
Property P29	**Formula** A[] (MASTER_BITBUS.attempt_unlink <= 3) **Function** Verifies that the unlink attempt counter (attempt_unlink) never exceeds 3. Guarantees an upper bound on the number of disconnection attempts, ensuring deterministic termination even in case of slave non-response.	**FSM→C Utility** Bounds unlink attempts: maximum 3 UNLINK attempts. Code: if (++attempt_unlink > 3) { force_disconnect(); } else { resend_unlink_request(); }. Allows forced disconnection after 3 failures, avoiding the master remaining indefinitely waiting for unlink confirmation.
D.6 Sequencing with SDLC Layer		
Property P30	**Formula** A[] (MASTER_BITBUS.SEND_LINK_REQ imply MASTER_SDLC.slave_state_in_m == NRM) **Function** Verifies that each time the Bitbus master sends a link request, the SDLC state of the slave as seen by the master is NRM (Normal Response Mode). Guarantees that Bitbus link operations are only initiated when the SDLC layer is in a stable and operational state.	**FSM→C Utility** Imposes an SDLC precondition: before sending LINK_REQ, verify that slave_state == NRM. Code: if (slave_sdlc_state != NRM) { wait_for_sdlc_ready(); } send_link_request();. Ensures protocol layer coherence and avoids sending Bitbus requests on a non-established SDLC link.
Property P31	**Formula** A[] (MASTER_BITBUS.SEND_ALPHANUM_M imply MASTER_SDLC.slave_state_in_m == NRM) **Function** Verifies that each time the master sends alphanumeric data, the SDLC state is NRM. Guarantees that application data exchanges only occur on an established and stable SDLC connection.	**FSM→C Utility** Precondition for data sending: verify slave_state == NRM before each data transmission. Code: assert(slave_sdlc_state == NRM); send_alphanum_data();. Prevents data loss by ensuring that the link layer is operational.
Property P32	**Formula** A[] (MASTER_BITBUS.SEND_UNLINK_REQ imply MASTER_SDLC.slave_state_in_m == NRM) **Function** Verifies that each time the master sends an unlink request, the SDLC state is NRM. Guarantees that	**FSM→C Utility** Precondition for unlinking: SDLC must be in NRM. Code: if (slave_sdlc_state == NRM) { send_unlink_request(); } else { log_error("SDLC not ready"); }.

	unlink operations are performed in the context of an active SDLC connection.	Ensures clean session termination on an active link.
Property P33	**Formula** A[] (MASTER_BITBUS.SEND_ALPHANUM_M imply !SLAVE_NO_LINKED) **Function** Verifies that each time the master sends alphanumeric data, the slave is linked (SLAVE_NO_LINKED == 0). Guarantees that no application data exchange occurs without a previously established Bitbus link.	**FSM→C Utility** Double precondition: SDLC in NRM AND Bitbus linked. Code: if (slave_sdlc_state == NRM && slave_linked) { send_data(); } else { error_not_linked(); }. Reinforces security by verifying both protocol layers before data transmission.
Property P34	**Formula** A[] (MASTER_BITBUS.SEND_UNLINK_REQ imply !SLAVE_NO_LINKED) **Function** Verifies that each time the master sends an unlink request, the slave is already linked. Guarantees logical coherence by prohibiting unlink attempts on a non-existent connection.	**FSM→C Utility** Avoids invalid UNLINKs: only unlink if already linked. Code: if (!slave_linked) { log_error("Already unlinked"); return; } send_unlink_request();. Prevents logic errors and inconsistent states.
D.7 Error Handling and Reset		
Property P35	**Formula** MASTER_BITBUS.ERROR_IN_M --> MASTER_BITBUS.SDLC_CONNECTION_REQUEST **Function** Verifies that each time the master detects an error (ERROR_IN_M), it will eventually reach the reset state (SDLC_CONNECTION_REQUEST). Guarantees that any error detected on the master side leads to controlled protocol reinitialization.	**FSM→C Utility** Error recovery strategy: any master error triggers complete reset to SDLC_CONNECTION_REQUEST. Code: if (error_detected) { cleanup_state(); state = SDLC_CONNECTION_REQUEST; }. Simplifies error handling by using a global reset strategy rather than complex local recoveries.
Property P36	**Formula** SLAVE_BITBUS.ERROR_IN_S --> MASTER_BITBUS.SDLC_CONNECTION_REQUEST **Function** Verifies that each time the slave detects an error (ERROR_IN_S), the master will eventually reach the reset state. Guarantees propagation of slave errors to the master and synchronization of error recovery.	**FSM→C Utility** Master-slave synchronization in error: a slave error must be signaled to the master which initiates the reset. Implies an error signaling mechanism (flag, special message). Master code: if (slave_error_received) { state = SDLC_CONNECTION_REQUEST; }. Ensures coordinated recovery of both parties.
Property P37	**Formula** A[] (MASTER_BITBUS.ERROR_IN_M imply MASTER_SDLC.slave_state_in_m == NRM) **Function** Verifies that each time the master is in error state, the underlying SDLC state is NRM. Guarantees that	**FSM→C Utility** Distinguishes Bitbus vs SDLC errors: a Bitbus error occurs on operational SDLC. If SDLC is not in NRM, the error is not Bitbus but SDLC. Guides diagnosis: if (error

	Bitbus errors occur only in the context of an established SDLC connection, excluding errors due to a failing link layer.	&& sdlc_state != NRM) { root_cause = SDLC_LAYER; } else { root_cause = BITBUS_LAYER; }.
Property P38	**Formula** A[] (SLAVE_BITBUS.ERROR_IN_S imply SLAVE_SDLC.slave_state == NRM) **Function** Verifies that each time the slave is in error state, the slave's SDLC state is NRM. Guarantees the same coherence on slave side: application errors are detected on a functional link layer.	**FSM→C Utility** Same principle on slave side: Bitbus error only if SDLC in NRM. Simplifies error diagnosis and fault localization in the layered architecture.
D.8 Reachability of Key States		
Property P39	**Formula** E<> MASTER_BITBUS.PROC_SEND_LINK_REQ **Function** Verifies the reachability of the state where the master processes sending a link request. Confirms that the link phase is effectively accessible in the model, validating completeness of the connection establishment scenario.	**FSM→C Utility** Validates link code completeness: confirms that the path to PROC_SEND_LINK_REQ exists. Justifies connection establishment tests and guarantees that this functionality is implemented and accessible.
Property P40	**Formula** E<> MASTER_BITBUS.PROC_SEND_UNLINK_REQ **Function** Verifies the reachability of the state where the master processes sending an unlink request. Confirms that the graceful connection termination phase is modeled and accessible.	**FSM→C Utility** Validates unlink code completeness: confirms that the graceful disconnection path is implemented. Guides session termination and resource cleanup tests.
Property P41	**Formula** E<> MASTER_BITBUS.PROC_SEND_ALPHANUM_M **Function** Verifies the reachability of the state where the master processes sending alphanumeric data. Confirms that the application data exchange phase is accessible, validating the nominal communication scenario.	**FSM→C Utility** Validates complete nominal path: from initialization to application data sending. Confirms that implementation allows achieving the protocol's main objective (data transfer).
Property P42	**Formula** E<> SLAVE_BITBUS.SEND_LINK_RESP **Function** Verifies the reachability of the state where the slave sends a link response. Confirms that the slave can respond to link requests, validating reactive behavior on slave side for connection establishment.	**FSM→C Utility** Validates slave reactivity: confirms that the slave can process and respond to LINK_REQ. Guides bidirectional handshake and master-slave synchronization tests.
Property P43	**Formula** E<> SLAVE_BITBUS.SEND_UNLINK_RESP **Function**	**FSM→C Utility** Validates bidirectional cleanup: confirms that the slave actively

	Verifies the reachability of the state where the slave sends an unlink response. Confirms that the slave can participate in graceful connection termination.	participates in termination. Guides clean disconnection and slave-side resource release tests.
Property P44	**Formula** E<> SLAVE_BITBUS.SEND_ALPHANUM_S **Function** Verifies the reachability of the state where the slave sends alphanumeric data. Confirms that the slave can emit data responses, validating complete bidirectional communication.	**FSM→C Utility** Validates bidirectional communication: confirms that the slave can not only receive but also send data. Justifies full-duplex communication and bidirectional data transfer tests.
D.9 Logical Coherence, Mutual Exclusion and Traceability		
Property P45	**Formula** A[] (MASTER_BITBUS.frame_sent_m == data_response_m imply SLAVE_NO_LINKED == 0) **Function** Verifies that each time a data response type frame is sent, the slave is linked (SLAVE_NO_LINKED == 0). Guarantees that data responses are only issued in the context of an established Bitbus session.	**FSM→C Utility** Session invariant: never send data without active link. Code: assert(slave_linked == true); send_data_response();. Prevents logic errors where data would be sent outside session context.
Property P46	**Formula** A[] (MASTER_BITBUS.SEND_LINK_REQ imply SLAVE_NO_LINKED == 1) **Function** Verifies that each time the master sends a link request, the slave is not linked (SLAVE_NO_LINKED == 1). Guarantees logical coherence by prohibiting redundant link requests on an already established session.	**FSM→C Utility** Avoids redundant LINKs: only link if not already linked. Code: if (slave_linked) { log_error("Already linked"); return; } send_link_request();. Prevents state inconsistencies and optimizes by avoiding unnecessary operations.
Property P47	**Formula** A[] (MASTER_BITBUS.SEND_UNLINK_REQ imply SLAVE_NO_LINKED == 0) **Function** Verifies that each time the master sends an unlink request, the slave is linked. Guarantees that only existing connections are unlinked, maintaining session state coherence.	**FSM→C Utility** Unlink coherence: only unlink if linked (see P34). Reinforcement with strict precondition in code.
Property P48	**Formula** MASTER_BITBUS.ERROR_IN_M --> MASTER_BITBUS.PROCESS_BITBUS **Function** Verifies that each time the master detects an error, it will eventually return to the PROCESS_BITBUS processing state. Guarantees that after error handling and reset, the system returns to a stable operational state allowing protocol resumption.	**FSM→C Utility** Guarantees recovery: after error and reset, return to operational state (PROCESS_BITBUS). Ensures that code does not remain blocked in permanent error state. Recovery structure: error_handler() → reset() → state = PROCESS_BITBUS; resume_normal_operation();

Property P49	**Formula** A[] not (MASTER_BITBUS.SEND_LINK_REQ and MASTER_BITBUS.SEND_UNLINK_REQ) **Function** Verifies strict mutual exclusion between sending link and unlink requests. Guarantees that no system state allows these two contradictory operations simultaneously.	**FSM→C Utility** LINK/UNLINK mutual exclusion: implementation via mutually exclusive states. Code: enum State { SEND_LINK_REQ, SEND_UNLINK_REQ, ... }; with a single state variable, naturally guarantees mutual exclusion. No need for additional locks.
Property P50	**Formula** A[] not (MASTER_BITBUS.SEND_ALPHANUM_M and MASTER_BITBUS.SEND_LINK_REQ) **Function** Verifies mutual exclusion between sending alphanumeric data and link requests. Guarantees that link and data transfer phases do not overlap.	**FSM→C Utility** LINK then DATA sequencing: never simultaneous. Strict sequential implementation: complete LINK before authorizing DATA. Code: state = SEND_LINK_REQ; wait_link_complete(); state = SEND_ALPHANUM_M;
Property P51	**Formula** A[] not (MASTER_BITBUS.SEND_ALPHANUM_M and MASTER_BITBUS.SEND_UNLINK_REQ) **Function** Verifies mutual exclusion between sending alphanumeric data and unlink requests. Guarantees that data transfers and session termination are sequential and not concurrent.	**FSM→C Utility** DATA then UNLINK sequencing: complete transfers before unlinking. Code: finish_data_transfer(); state = SEND_UNLINK_REQ;. Avoids data loss in transit during disconnection.
Property P52	**Formula** A[] not ((SLAVE_BITBUS.SEND_LINK_RESP and SLAVE_BITBUS.SEND_UNLINK_RESP) or (SLAVE_BITBUS.SEND_LINK_RESP and SLAVE_BITBUS.SEND_DATA_RESPONSE) or (SLAVE_BITBUS.SEND_UNLINK_RESP and SLAVE_BITBUS.SEND_DATA_RESPONSE)) **Function** Verifies complete mutual exclusion between the three types of slave responses (link, unlink, data). Guarantees that the slave sends only one type of response at a time, ensuring exchange coherence and non-ambiguity.	**FSM→C Utility** Ternary mutual exclusion: only one response type at a time. Implementation: enum ResponseType { LINK_RESP, UNLINK_RESP, DATA_RESP }; ResponseType current_response;. Simplifies master reception code which only has one response type to handle per cycle.
Property P53	**Formula** MASTER_BITBUS.SEND_LINK_REQ --> (MASTER_BITBUS.RECEIVE_LINK_RESP or MASTER_BITBUS.ERROR_IN_M or MASTER_BITBUS.SDLC_CONNECTION_REQUEST) **Function** Verifies request-response traceability for linking: after sending a link request, the system will reach either response reception, or an error state, or a new SDLC	**FSM→C Utility** Guarantees deterministic future after LINK_REQ: three clearly defined possible outcomes. Guides timeout and error handling implementation: after SEND_LINK_REQ, wait for RECEIVE_LINK_RESP with timeout leading to ERROR_IN_M or SDLC_CONNECTION_REQUEST.

	connection request. Guarantees that no link request remains without a defined outcome.	Code: send_link_req(); wait_response_or_timeout(); handle_outcome();
Property P54	**Formula** MASTER_BITBUS.SEND_UNLINK_REQ --> (MASTER_BITBUS.RECEIVE_UNLINK_RESP or MASTER_BITBUS.SDLC_CONNECTION_REQUEST) **Function** Verifies request-response traceability for unlinking: after sending an unlink request, the system will reach either reception of an unlink response, or a new SDLC connection request. Guarantees that no unlink request leaves the system in an undefined state.	**FSM→C Utility** Guarantees two outcomes after UNLINK_REQ: response or reset. Implementation: send_unlink_req(); wait_response_or_timeout(); if (timeout) { state = SDLC_CONNECTION_REQUEST; } else { state = RECEIVE_UNLINK_RESP; }. Ensures clean termination even without slave confirmation.

Conclusion

The fifty-four formal properties presented in this appendix constitute a structured framework for verification of the Bitbus protocol. Each is associated with an operational interpretation that makes explicit its impact on embedded C implementation, thus establishing a methodological link between model checking verification and software development.
Systematic analysis of these properties is not limited to theoretical model validation. It allows explicit identification of functional and temporal constraints that must be respected during implementation, particularly regarding timeout management, recovery mechanisms, exclusive state transitions and error situation handling.
The whole fits into a controlled development approach for critical systems, where assertions derived from the formal model serve as a reference for structuring and constraining software implementation. This continuity contributes to strengthening traceability between specification, formal verification and software realization.