On TorXakis Correctness as an ioco Implementation: An Empirical Model-Based Evaluation

by

Reza Ghasemi

A thesis submitted to the
Department of Computer Science
in conformity with the requirements for
the degree of Master of Science

Bishop's University
Canada
May 2025

Abstract

The ultimate goal of this thesis is to work toward the use of model-based testing techniques to evaluate whether TorXakis is a correct implementation of the input-output conformance (ioco) testing theory. Rather than pursuing a formal proof of correctness, we adopt an empirical, model-based testing approach to evaluate whether TorXakis adheres to the expected semantics defined by ioco. A wide range of custom test models were designed and executed, each targeting specific system behaviors such as concurrency, synchronization, fault tolerance, and deadlock handling. These models simulate real-world challenges to assess whether TorXakis produces outputs and traces that align with the theoretical behavior prescribed by ioco. Through structural, behavioral, and trace-based analysis, we collect evidence that supports or challenges TorXakis's conformance to ioco principles. The tests were conducted under controlled conditions, systematically increasing complexity to expose potential deviations or inconsistencies. While the study does not offer a formal verification of TorXakis, it provides a practical and meaningful evaluation that lays the groundwork for future formal investigations.

Acknowledgments

I want to express my heartfelt gratitude to my supervisor, Dr. Bruda, for his invaluable guidance, constructive feedback, and continuous encouragement throughout my research. His expertise and insights have been instrumental in shaping this thesis. The feedback of the members of my examining committee (Dr. Butler, Dr. Malik, and Dr. Hedjam) resulted in significant improvements of this manuscript.

I deeply thank my family and friends for their unwavering support and understanding during this journey. Their belief in my abilities provided me with the strength to persevere.

Lastly, I extend my appreciation to the developers and contributors of TorXakis for creating such a robust tool, which served as the foundation for my work.

Contents

Chapter 1

Introduction

Software testing is a critical aspect of software development, aimed at identifying defects, errors, and vulnerabilities in software systems to ensure their quality and reliability. Fundamental concepts in software testing include test case generation, test execution, defect detection, and test coverage analysis.

Program verification is the process of ensuring that a software program satisfies specified requirements or properties. It involves analyzing the program's source code or executable to determine whether it behaves as intended and does not exhibit any undesired behaviors, such as errors, bugs, or security vulnerabilities. Verification techniques may include formal methods, static analysis, dynamic analysis, testing, and theorem proving [6, 29].

Model-based testing is a software testing technique that involves creating abstract models of the system under test (SUT) and generating test cases from these models. These models capture the behavior, structure, and interactions of the SUT, allowing testers to systematically derive test cases that cover various scenarios and functionalities. Model-based testing can improve test coverage, reduce testing time and effort, and enhance the effectiveness of software validation processes [8, 17].

Model-based testing (MBT) is grounded in the principle that a system's expected behavior can be described using a formal model, from which test cases can be systematically derived and executed. This approach enables exhaustive test coverage and allows for early defect detection, making it a powerful technique for validating complex software systems. Several foundational works have contributed to the development of MBT methodologies, including the seminal work by Utting [18], which provides a comprehensive overview of MBT techniques, tools, and applications.

A key aspect of MBT is the formal representation of system behavior using modeling formalisms such as finite state machines (FSM), labeled transition systems (LTS), and process algebra. These models serve as a basis for generating test cases that exercise various paths, transitions, and interactions within the system. Furthermore, MBT can be classified into offline and online testing approaches. In

offline MBT, test cases are generated prior to execution and stored for later use, whereas in online MBT, test cases are generated dynamically during execution based on the system's response. The latter approach provides greater adaptability in testing real-time and adaptive systems [2].

TorXakis in particular is a toolset for model-based testing and verification of concurrent systems. It provides a formal specification language for describing system behaviors and properties, as well as a set of tools for model simulation, test generation, and verification [32]. TorXakis is an implementation of input-output conformance (ioco) [36], a formal framework used to assess whether the observable behavior of an implementation conforms to the expected behavior defined by its specification. TorXakis automates the process of test generation and application, based on a specification formulated using a process algebra.

1.1 Toward a Formal Verification of TorXakis

Model-based testing (MBT) in general, and TorXakis in particular, have been successfully applied to validate real-world software systems such as network protocols, Web applications, and client-server systems, and even in an academic testing as a grading tool [1, 12, 37]. According to ioco, this establishes formally the correctness of the respective applications. However, there is one big elephant in the room. Indeed, there is no verification of the verification tool itself, and so the formal correctness argument relies on the unverified hypothesis that TorXakis itself is a correct implementation of ioco.

The fundamental question that we start addressing in this paper is whether TorXakis is a correct implementation of ioco. Conceivably we can use another formal verification tool for this purpose, but the question of the correctness of that tool appears recursively with no end in sight. Out thesis is therefore that instead of relying on some third part tool TorXakis can be verified using model-based testing that is, using TorXakis itself.

It should be noted that TorXakis does come with a testbench. However, that testbench is limited to six token tests which are more a proof of concept rather than any guarantee of correctness, and not even near real world scenarios of modern software testing environments [36, 38]. While TorXakis has been widely adopted for its effectiveness in validating software systems, the limited scope and simplicity of its default test suite remain a significant drawback [11, 15].

In this paper we investigate extensions of the TorXakis testbench. We do this by designing a wide variety of test models that reflect different aspects of ioco semantics—such as observable actions, non-deterministic behaviors, refusals, quiescence, and deadlock scenarios—and running these models through the TorXakis testbench. By analyzing the outcomes, we examine whether the tool behaves in a manner consistent with what ioco would theoretically predict.

Our contribution is two-fold. On one hand, we establish a practical and structured framework for testing TorXakis, offering empirical evidence that demonstrates the robustness of TorXakis in the context of testing modern, complex software. We validate TorXakis across a broader spectrum of tests, including concurrency stress, fault tolerance, and scalability. Testing the tool itself is crucial for ensuring its reliability, accuracy, and effectiveness. Proper validation allows developers to identify and resolve potential bugs, glitches, or functional limitations before the tool is employed in real-world software testing [32, 36]. The quality of a testing tool directly impacts its usability and reliability; if it is not thoroughly tested, it may produce inaccurate results, overlook critical issues, or fail to handle complex scenarios effectively. This could lead to unreliable software testing outcomes and potentially compromise the quality and stability of the tested software [24, 30]. In other words, rigorous testing of the testbench itself is an essential step in the software development process to ensure that it performs as expected and can effectively support software validation across diverse scenarios [38].

On the other hand, we try to establish a basis for the formal verification of TorXakis using TorXakis itself. Once all out tests are run, we aim to draw some however tentative conclusions about the correctness of TorXakis. We also aim to determine whether TorXakis is powerful enough to handle the more complex testing scenarios that such a formal self-verification will most likely entail.

Chapter 2

Previous Work

2.1 Formal Models

A **Labeled Transition System (LTS)** is a foundational formalism used to describe the behavior of concurrent and reactive systems. It provides a structured mathematical framework that models the possible execution paths of a system through states and labeled transitions [16, 36].

Formally, an LTS is defined as a triple $(S, Act, \rightarrow)$, where S is a finite set of states, Act is a set of observable actions (also known as labels), and $\rightarrow \subseteq S \times Act \times S$ is a transition relation. A transition of the form (s_1, a, s_2), often written as $s_1 \xrightarrow{a} s_2$, denotes that the system, when in state s_1, can perform action a and move to state s_2.

A trace is defined as a finite sequence of observable actions that represents a possible execution path of the system from its initial state. Formally, a trace is a sequence $t = a_1, a_2, \ldots, a_n$, where each action a_i belongs to the set Act, and there exists a corresponding sequence of states $s_0, s_1, \ldots, s_n$ such that:

$$s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} s_2 \ldots \xrightarrow{a_n} s_n$$

The set of all such traces characterizes the external behavior of the system and defines what is observable from its execution. To illustrate, consider a simple vending machine modeled as a Labeled Transition System (LTS). The machine begins in a state labeled *Start*, moves to *Paid* upon receiving a coin through the action `insertCoin`, and then transitions to *Dispensed* when the user presses the button via the action `pressButton`.

The corresponding transitions can be written as:

$$Start \xrightarrow{\texttt{insertCoin}} Paid \xrightarrow{\texttt{pressButton}} Dispensed$$

A valid trace for this system is:

```
insertCoin, pressButton
```

This reflects a sequence of observable interactions that leads to successful product delivery. This formalism serves as the foundation for analyzing, verifying, and testing system behavior in model-based testing frameworks such as TorXakis.

LTS are typically described suing a *process algebra*, which is a mathematical framework designed to model and reason about the behavior of concurrent systems. It provides a formal language and set of operators that describe how processes behave, interact, and evolve over time through observable actions. Each process in this framework represents an abstract behavior that can engage in communication, make decisions, perform sequences of actions, or operate in parallel with other processes. [5]

There are many process algebra such as Calculus of communicating systems (CCS) [19] and Communicating sequential processes (CSP) [26]. TorXakis uses yet another process algebraic notation to describe specifications, which we will also use here for the sake of consistency.

In this formalism, an action prefixing operator defines the fundamental structure of a process: for instance, the expression a >-> P represents a process that performs action a and then continues as process P. This enables a natural way to describe sequential behaviors. Beyond sequentiality, systems often need to exhibit nondeterministic behavior, where multiple alternatives are possible. The operator P ## Q expresses a nondeterministic choice between processes P and Q, allowing the system to proceed along either path based on internal or environmental conditions.

Concurrency is another key aspect modeled in process algebra. The operator P ||| Q describes two processes executing concurrently without synchronization, meaning that their actions may interleave freely. When synchronization is required on specific actions, the operator P |[A]| Q is used. This ensures that both processes must agree to simultaneously perform any action belonging to the set A, while other actions remain independent. Such synchronization mechanisms allow the modeling of tightly coordinated systems, such as protocols or resource-sharing architectures.

In some scenarios, abstraction is necessary to simplify internal details of a process. This is achieved through action hiding, where certain internal actions are rendered unobservable to external observers. Though TorXakis does not use an explicit hide keyword in the core syntax, similar effects can be achieved by restricting or renaming channels during process composition and specification.

Recursive definitions are also integral to process algebra, as they allow the modeling of behaviors that repeat indefinitely. For example, the TorXakis definition PROCDEF counter [Tick] () ::= Tick >-> counter [Tick] () ENDDEF models a process that performs the action Tick repeatedly without termination. This style of definition is essential for representing loops, timers, or reactive systems that continuously respond to external stimuli.

Taken together, these operators enable the construction of complex and realistic models of system behavior. In TorXakis, process algebra serves as the foundation

for defining formal models that are both analyzable and executable. Its expressive power allows system designers and testers to describe not only the nominal behavior of systems, but also exceptional scenarios such as deadlocks, race conditions, and synchronization faults. Throughout this thesis, these constructs are applied extensively to model, test, and evaluate systems under varying conditions of stress, fault, and concurrency.

Test generation in model-based testing (MBT) refers to the automated derivation of test cases from formal models that describe the expected behavior of a system under test. Rather than writing tests manually, MBT leverages a behavioral specification—such as a labeled transition system (LTS) or process algebra—to systematically produce test cases that are both comprehensive and formally grounded. This approach ensures that the generated tests are consistent with the model and can cover a wide range of scenarios, including normal execution paths, edge cases, and potential faults. MBT enables the automation of test case generation by interpreting the structure and semantics of the model to drive the construction of test inputs and expected outputs. This paradigm is especially valuable for complex systems, where manually designing an adequate set of tests is not only time-consuming but also error-prone. In this thesis, test generation is carried out using TorXakis, a model-based testing tool that relies on formal models to produce and execute test sequences against the system specification [18].

2.2 Input-Output Conformance (ioco) Theory

Input-Output Conformance (ioco) is a formal theory used to assess whether the observable behavior of an implementation conforms to the expected behavior defined by its specification. It is widely used in model-based testing, particularly for systems that interact with their environment through inputs and outputs. The ioco framework is grounded in the use of Labelled Transition Systems (LTS), which describe the behavior of a system in terms of its states and transitions, with each transition being labeled as either an input or an output action [36].

Under the ioco framework, a system implementation is said to conform to its specification if, after any sequence of inputs and outputs allowed by the specification, the outputs produced by the implementation are also allowed by the specification. This includes the concept of quiescence, which represents the absence of output. Quiescence is treated as an observable behavior in ioco, meaning that the system's inaction in certain contexts must also conform to the specification.

The importance of ioco in model-based testing lies in its ability to provide a systematic and rigorous foundation for conformance checking. Instead of relying on manually written test cases, ioco enables the automatic generation of test cases from a formal specification, ensuring thorough coverage of the system's behavior. This is particularly effective for reactive systems, where the system continuously responds to external stimuli [36].

The Input-Output Conformance (ioco) relation is formally defined based on labeled transition systems (LTS) with inputs and outputs. Given a specification *spec* and an implementation *impl*, both modeled as input-output labeled transition systems, the ioco relation is defined as follows:

$$\textit{impl ioco spec} \iff \forall s \in \text{traces(spec)} : \text{out}(\textit{impl after } s) \subseteq \text{out}(\textit{spec after } s)$$

In this context, traces(*spec*) denotes the set of all observable traces—that is, sequences of actions that the specification can perform. The expression *out*(p after s) refers to the set of outputs that a process p can produce after executing the trace s, including quiescence, which is typically denoted by the symbol δ. The terms *impl after s* and *spec after s* refer to the reachable states of the implementation and specification, respectively, after the execution of trace s.

This definition ensures that the implementation never produces outputs that the specification does not allow after any valid interaction sequence. In particular, it also captures situations where the system becomes quiescent (i.e., does not produce any output), treating such behavior as a significant observable event. IOCO provides a strong and mathematically grounded foundation for conformance testing in model-based verification frameworks.

The ioco framework is highly relevant to model-based testing (MBT) because it provides a rigorous definition of behavioral conformance between a system implementation and its formal specification. In MBT, test cases are derived from a model that describes the expected behavior of the system under test. The ioco relation defines what it means for an implementation to conform to this model, ensuring that after any sequence of inputs, the implementation does not produce outputs that are disallowed by the specification. This makes ioco a powerful foundation for automatic test generation, verdict calculation, and test oracle design in MBT [36].

TorXakis uses labeled transition systems and process algebra to describe system behavior and generate test cases accordingly. Its ability to simulate and validate observable outputs after traces mirrors the essence of ioco conformance. Thus, ioco can be seen as a conceptual backbone that justifies and strengthens the model-based validation capabilities provided by TorXakis.

TorXakis implements the ioco testing theory through a series of integrated mechanisms. First, it models systems as Labeled Transition Systems (LTS), derived from formal process algebra specifications. It then performs on-the-fly trace exploration to dynamically analyze system behavior as traces are executed. During this exploration, TorXakis checks whether the outputs produced by the implementation after a given trace are included within the outputs allowed by the specification. Additionally, the tool supports automated test generation under the ioco framework, enabling efficient conformance testing of reactive systems based on formally defined behavioral models [33].

2.3 Related Work and Current State of Research

Model-based testing (MBT) as a formal verification technique has a long history grounded in the definition of various preorder implementation relations [9], which in turn allow the algorithmic generation of sound and complete tests suites [39]. Modern practical MBT is based on I/O transition systems [7]. Seminal contributions by Tretmans have shaped the theoretical foundation of MBT, particularly through the development of the input-output conformance (ioco) testing framework [35, 36]. His work introduces the use of labeled transition systems (LTS) to model the expected behavior of systems and defines conformance relations that can be used to verify whether implementations meet their specifications. Later extensions further established the soundness and exhaustiveness of MBT when applied under well-defined modeling assumptions.

Building upon these theoretical foundations, recent research has turned toward enhancing the practical capabilities of MBT tools, particularly their testbenches. An analysis of existing literature, including the works of Pretschner el. al. and Forgacs, reveals a significant gap in testbench capabilities for model-based testing tools, particularly in autonomous operations and extensive test case generation [10, 23].

Pretschner el. al. focus on the development of model-based testing techniques for complex software systems. They emphasize the importance of automated test case generation and highlight the limitations of current testbenches in this regard. Forgacs, on the other hand, explores the role of autonomy in software testing, advocating for the integration of intelligent algorithms into testbench frameworks to enable autonomous operations.

Despite their valuable contributions, both Pretschner el. al. and Forgacs identify the need for further advancements in testbench capabilities to address the evolving challenges of software validation. Specifically, there is a consensus on the necessity for testbenches that can autonomously generate diverse and extensive test cases to ensure comprehensive coverage of software functionalities and behaviors [10, 23].

Alternatives to Model-Based Testing

In the domain of software testing, there are various methods beyond model-based testing that offer different perspectives and approaches to ensure software quality. Techniques like static analysis, fuzz testing, and mutation testing provide additional ways to strengthen software quality assurance efforts [4, 20, 21]. Static analysis involves carefully examining software code without actually running it, aiming to uncover any flaws or vulnerabilities. Fuzz testing, on the other hand, involves bombarding a software system with unconventional or invalid inputs to uncover hidden bugs or security weaknesses [14]. Mutation testing, meanwhile, introduces small changes to the source code and then evaluates how well existing test suites detect these alterations, giving insight into the effectiveness of the testing approach cite [22, 27].

While model-based testing remains popular, it's important to explore other methods that bring unique benefits and perspectives. Exploratory testing, for example, encourages testers to spontaneously explore a software application to uncover defects and assess its usability, relying on their expertise and creativity[31]. Behavior-driven development (BDD) focuses on expressing test scenarios in natural language to align with stakeholder expectations and promote collaboration between different parties involved in software development[13]. Each of these alternatives comes with its own advantages and challenges, emphasizing the need for a broad understanding of different testing methodologies to ensure comprehensive software quality assurance.

The current state of research in the field of model-based testing reflects a growing interest in enhancing testbench capabilities to meet the demands of modern software development [25]. Recent studies have proposed various approaches to address the limitations identified by Pretschner el. al. and Forgacs, including the integration of machine learning algorithms, the development of domain-specific modeling languages, and the adoption of cloud-based testing platforms [3, 25].

However, despite these efforts, there remains a gap between the theoretical advancements proposed in the literature and their practical implementation in existing testbench frameworks [32]. Many of the proposed solutions are still in the experimental stage and have not been widely adopted by industry practitioners.

2.4 Performance Evaluation Tests in TorXakis

The tests listed in benchtest focused on evaluating the performance characteristics of different types of processes and constructs in TorXakis. Let's break down each test and provide more details [34]:

Test Performance of Sequence
This test evaluates the performance of sequential processes in TorXakis. It involves measuring the time taken to execute a sequence of actions or the throughput of sequential processes. Sequential processes are fundamental in modeling systems where actions occur one after another in a predetermined order. The test involves scenarios where multiple sequences are executed concurrently or in different configurations to assess performance under various conditions.

Test Performance of Synchronized Processes
This test focuses on evaluating the performance of processes that synchronize on certain events or conditions. Synchronized processes typically coordinate their actions based on shared variables, signals, or message passing. The test involves measuring the latency and overhead associated with synchronization mechanisms

such as barriers, semaphores, or message queues. Evaluating how efficiently TorXakis handles synchronization can be crucial for modeling systems with concurrent or distributed components.

Test Performance of Parallel Processes
This test aims to assess the performance characteristics of parallel processes in TorXakis. Parallel processes execute simultaneously and may interact with each other through shared resources or communication channels. The test involves measuring factors such as scalability, resource utilization, and potential bottlenecks in executing parallel processes. It explores scenarios where the number of parallel processes varies or where workload distribution strategies are employed.

Test Performance of Hidden Processes
Hidden processes are processes that do not directly participate in the main behavior of a system but play a supporting role, such as performing background tasks or managing resources. This test evaluates the performance impact of hidden processes on the overall system behavior. It involves measuring the overhead introduced by hidden processes, as well as their effectiveness in supporting the primary functionality of the system. Evaluating hidden processes is important for understanding their influence on system performance and identifying potential optimization opportunities.

Nesting
Nesting refers to the composition of processes within other processes, allowing for hierarchical modeling and abstraction. This test examines the performance implications of nesting processes to represent complex systems. It involves measuring the overhead of managing nested processes, as well as assessing the scalability and maintainability of nested models. Evaluating nesting capabilities helps determine the suitability of TorXakis for modeling systems with varying levels of complexity and abstraction.

Test Performance of Enable
The "enable" construct in TorXakis specifies conditions under which certain actions or processes become enabled or disabled. This test evaluates the performance of enable conditions in controlling the execution flow of processes. It involves measuring the responsiveness and efficiency of enable conditions, as well as their impact on overall system behavior. Evaluating enable conditions helps ensure that TorXakis effectively captures the dynamic nature of systems where actions are contingent on specific conditions.

These tests collectively aim to assess the performance characteristics, scalability, and efficiency of TorXakis in modeling and analyzing concurrent and distributed

systems. By conducting these tests, users can gain insights into how well TorXakis handles various modeling scenarios and identify areas for improvement or optimization.

2.5 Additional Empirical Tests for Evaluating TorXakis's Conformance to ioco

In addition to the existing tests outlined in benchtest, we propose incorporating further tests to comprehensively evaluate different aspects of TorXakis and to focus on specific features or scenarios relevant to our use case. These additional tests aim to provide a strong examination of TorXakis's capabilities and behavior under various conditions.

Concurrency stress test involves creating a highly concurrent workload to stress test TorXakis's ability to handle a large number of concurrent processes. It involves scenarios where a significant number of processes are executing simultaneously, with frequent interactions and synchronization points.

Fault tolerance test evaluates TorXakis's behavior in the presence of faults or errors, such as process failures, message loss, or communication disruptions. It involves injecting faults into the system and observing how TorXakis reacts, recovers, or maintains system integrity.

Scalability test assesses TorXakis's scalability by increasing the size or complexity of the modeled system and measuring its performance. It involves gradually increasing the number of processes, channels, or interactions in the model and observing how TorXakis handles the increased workload.

Resource utilization test measures the resource utilization of TorXakis, such as CPU usage, memory consumption, and network bandwidth. It involves monitoring resource metrics while running various test scenarios to identify potential resource bottlenecks or inefficiencies.

Model verification test focuses on verifying the correctness of models created with TorXakis by comparing their behavior against expected specifications or properties. It involves defining formal properties or assertions and using TorXakis to verify them against the model's behavior.

Integration test evaluates TorXakis's integration with other tools, frameworks, or environments. It involves integrating TorXakis with external systems or tools and verifying interoperability, data exchange, or compatibility.

Chapter 3

Performance Testing and System Analysis

The testing phase of this research was designed to evaluate the developed model comprehensively. A series of tests were performed to ensure the system's reliability, scalability, and user experience, under real-world conditions. This chapter outlines the tests thus conducted.

Our models are constructed under several formal assumptions and behavioral expectations. We assume non-blocking execution, where all sequences and data processes emit outputs on their channels continuously, without waiting for inputs or delays—effectively simulating an open-loop system. Synchronized processes are expected to engage in a fixed number of interleaved synchronization steps, assuming all participating processes are available. The test environment is considered isolated, without external inputs or blocking reads. A fairness assumption is adopted, wherein all processes are assumed to be scheduled fairly so that none is indefinitely denied access to required synchronization.

Finally, it is assumed that the system is well-formed, meaning all channel references are unique and correctly typed, avoiding naming conflicts or mismatches across concurrent modules.

3.1 Maximum Concurrency Stress Test

The primary goal of this test is to evaluate how the system handles high levels of concurrency. This includes measuring response times, throughput, and resource usage under peak load conditions. By stressing the system, we can also identify potential bottlenecks or performance degradation points. This test also helps ensure that the system remains stable and performs correctly under heavy concurrent load without crashes or unexpected behavior. Finally, the test should detect concurrency issues including deadlocks, race conditions, and other synchronization issues that

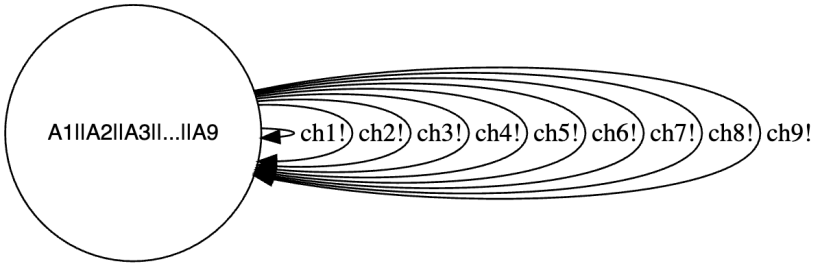

Figure 3.1: Simplified LTS Diagram for the Maximum Concurrency Stress Test

might only manifest under high concurrency.

The concurrency stress test is defined as a parallel composition of multiple processes, each communicating independently over its own dedicated channel. Each process repeatedly emits outputs over a distinct communication channel. Let A_i denote a process that continuously outputs on channel ch_i, defined as $A_i = ch_i! \rightarrow A_i$. The operator $\|$ indicates parallel composition, and $\|_{i=1}^{9} A_i$ represents the parallel execution of all nine such processes. The overall system model is thus expressed as follows:

$$\text{ConcurrencyModel} = A_1 \parallel A_2 \parallel A_3 \parallel A_4 \parallel A_5 \parallel A_6 \parallel A_7 \parallel A_8 \parallel A_9$$

where each process is defined as $A_i = ch_i! \rightarrow A_i$ for $i = 1, \ldots, 9$. This formal model represents an infinite loop of output actions from each channel, running concurrently, and constitutes the structural basis for testing the maximum parallelism that the system can support. In practice, however, the implemented test is significantly more complex and involves fourteen output channels to support a diverse set of concurrent communication patterns. These channels include basic output channels used for executing parallel sequences (Channel1 through Channel9), integer data channels for transmitting numerical values in parallel (ChannelInt1, ChannelInt2, and ChannelInt3), and structured data channels that handle sequences involving composite data types such as tuples or lists (Channel10Ints and Channel10Ints_b). All channels are unidirectional (output only) and are reused across multiple processes to simulate intense synchronization demands and contention for shared resources. The complete TorXakis code for this test is provided in Appendix A.1. The semantics of this test is given by the LTS shown in Figure 3.1.

By simulating a high-concurrency environment with synchronized outputs and interleaved execution sequences, this test expects the system to remain robust and responsive. Specifically, the system should avoid deadlocks during intense parallel activity, maintain consistent synchronization across interleaved steps, and support execution with up to 14 channels without failures or unexpected behavior. Faults such as starvation or deadlock are only expected to appear under extreme resource exhaustion or misconfigured synchronization conditions.

The developers of TorXakis have designed the system to handle up to 14 channels and a limited number of processes (typically 4 or 5) as part of their benchmark tests, which aim to evaluate scalability under expected usage scenarios. However, these limitations are not inherent to TorXakis but rather reflect the typical performance boundaries tested under standard configurations. In our tests we sought to exceed these predefined constraints by manually increasing the number of channels and processes.

While TorXakis can theoretically handle a larger number of channels and processes, practical constraints, such as computational resources and execution time, come into play. For example, during simulations with 14 channels our test machine required approximately 5 hours to complete the test. When we attempted to scale the system further, testing configurations with 100 channels and 50 processes, we encountered performance bottlenecks, particularly with memory usage and execution time. These resource limitations prevented the successful execution of simulations with more than 14 channels, highlighting the practical challenges associated with stress tests on personal computing hardware.

It is important to note that the limitations observed in these tests are primarily determined by the hardware's computational capacity rather than any strict constraints within TorXakis itself. Model-based testing tools like TorXakis are inherently resource-intensive, as they must explore all possible states and transitions in a model. When the number of channels and processes increases, the tool faces the critical challenge of *state space explosion*: TorXakis explores all possible states and transitions, and a larger number of channels and processes expands this state space beyond what typical personal computers can efficiently handle.

3.1.1 Deadlock Injection and Detection Evaluation

The core idea behind the intentional deadlock test in our TorXakis concurrency test is to create a controlled environment where deadlocks are intentionally introduced. To define such a scenario, each process waits for communication on a specific channel and then terminates. We then arrange these processes in a closed synchronization set, where each one depends on the next to proceed. The overall system creates a circular wait pattern that leads to a deadlock. We observe whether the system under test reaches a state with no outgoing transitions, a classical indicator of deadlock in LTS semantics.

The intentional deadlock model is designed by composing four atomic processes, each attempting to send an output on a distinct channel before halting. Formally, each process D_i is defined as $D_i = ch_i! \rightarrow STOP$, where $i = 1 \ldots 4$. These processes are then composed in parallel and synchronized over a set of shared channels to induce a circular dependency as follows:

$$\text{DeadlockModel} = (D_1 \parallel D_2 \parallel D_3 \parallel D_4) \setminus \{ch1, ch2, ch3, ch4\}$$

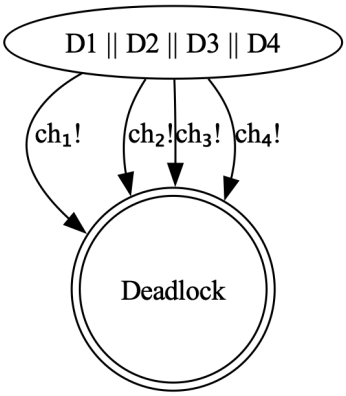

Figure 3.2: Simplified LTS Diagram for the Deadlock Injection

In this model (Figure 3.2), the synchronizations are defined such that D_1 and D_2 share channel $ch1$, D_2 and D_3 share channel $ch2$, D_3 and D_4 share channel $ch3$, and finally, D_4 and D_1 share channel $ch4$. This configuration forms a closed synchronization loop. The LTS illustrates this structure by showing each of the four processes emitting an output over their respective channel, all leading to a single deadlocked state. This algebraic model serves as a simplified representation of the actual test structure and is included here for clarity. In practice, the implemented test is more complex and aligns with the LTS diagram shown. The communication infrastructure supporting this model consists of fourteen output-only channels, each reused across multiple concurrent process definitions. These include basic deadlock channels such as Channel1 through Channel4 used in the deadlock sequences, synchronized interaction channels (e.g., Channel4 to Channel6), nested synchronization channels, parallel integer dataflow channels, and structured channels for complex message types. All of these channels are deliberately shared to simulate synchronization conflicts and circular dependencies. This full version introduces additional layers of concurrency and synchronization to more closely simulate real-world scenarios, extending beyond the minimal four-process abstraction. We thus create a common deadlock scenario called circular deadlock, which is among the most common deadlock scenarios in real-world concurrent systems.

This allows us to verify how the system handles situations where processes are unable to proceed due to circular dependencies on resources (channels). By setting up these intentional deadlocks, we can observe and analyze how TorXakis detects, reports, and potentially recovers from deadlock conditions. The system successfully handled the test cases using the code. See Appendix A.2 for full model code.

This test is representative because it addresses a fundamental cause of deadlocks—dependency cycles among resources. The standard suite for TorXakis includes basic deadlock scenarios, typically focused on simple two-process deadlocks or synchronization based deadlocks. These involve mutual exclusion or a simple chain of dependencies but may not extend to complex cyclic dependencies among multiple processes. Our test goes further by involving multiple channels and interprocess dependencies, creating a more complex circular wait situation. This pushes TorXakis to detect and handle deadlocks beyond what simple tests in the standard suite address, simulating a more realistic and complex deadlock scenario.

3.1.2 Real-World Scenario Coverage

Table 3.1 summarizes coverage percentages for various real-world concurrency scenarios based on common concurrency testing features and the expected effectiveness in detecting concurrency faults. The table includes the approximate coverage for each scenario, the types of concurrency faults it addresses, and the reasoning behind each coverage percentage.

The comparison presented in Table 3.1 provides a structured overview of how different concurrency test scenarios map to real-world fault detection capabilities. Each test type targets specific levels of system complexity, channel usage, and synchronization characteristics, offering insights into their respective strengths and limitations.

The *Basic Concurrency Test* shows limited coverage (20–30%) due to its minimal setup with only a few channels and sequential process flow. While sufficient for detecting elementary synchronization faults and deadlocks, it lacks the ability to model more realistic scenarios involving interaction timing or parallel execution.

The *Moderate Concurrency Test* improves coverage (50–60%) by introducing parallelism and a modest number of channels. This setup enables detection of basic race conditions in addition to deadlocks, making it a practical middle-ground for typical applications. However, it still lacks the scalability and dynamic timing variations required to fully emulate high-load systems.

As we progress to the *High-Concurrency Stress Test*, the use of 10–15 channels and more extensive parallelism increases the coverage to 70–80%. This test effectively reveals issues like resource starvation and complex interleavings but falls short in accounting for randomness in execution order, which can expose subtler concurrency bugs.

The two most advanced test types—*Dynamic Concurrency with Random Timing* and *Dynamic Concurrency with Complex Synchronization*—demonstrate the highest levels of expected coverage (85–90%). These scenarios model real-world environments more faithfully by incorporating nondeterministic timing and intricate synchronization logic. The former excels at capturing timing-sensitive faults such as race conditions through randomized execution patterns, while the latter focuses on

Scenario/Test Type	Key Features Included	Expected Coverage (%)	Types of Faults Detected	Reasoning
Basic Concurrency Test	Minimal channels (2-3), sequential processes	20-30%	Deadlocks, basic synchronization	Limited scope, low concurrency, does not address complex interactions or timing issues.
Moderate Concurrency Test	Medium channels (5-7), parallel processes	50-60%	Deadlocks, basic race conditions	Covers typical concurrency but lacks complexity and scalability for higher loads.
High-Concurrency Stress Test	Many channels (10-15), extensive parallelism	70-80%	Deadlocks, resource starvation	High concurrency but lacks intentional variations and random execution orders.
Dynamic Concurrency with Random Timing	Randomized execution timing, high channel count	85-90%	Deadlocks, race conditions	Captures timing-sensitive faults like race conditions; randomization aids in variability.
Dynamic Concurrency with Complex Sync	High channel count, complex synchronization	85-90%	Starvation, complex deadlocks	Extensive interaction patterns with complex syncs to cover resource and execution challenges.

Table 3.1: Comparison of Different Concurrency Tests

detecting starvation and deep deadlocks that arise from complex interdependencies between processes.

Overall, the comparison confirms that increasing model complexity and behavioral diversity through dynamic timing and synchronization significantly enhances fault coverage. However, this comes with the cost of increased system load, greater test execution time, and the need for more sophisticated modeling tools—factors that must be balanced based on the intended application domain and testing goals.

Our test is categorized as a High-Concurrency Stress Test, with an estimated coverage of 80-90%. It is designed to detect deadlocks, resource starvation, and some race conditions. By incorporating many channels and processes with recursion and parallel composition, our test achieves significant coverage and effectively simulates a high-concurrency environment, making it well-suited for identifying deadlocks

```
TXS >>  TorXakis :: Model-Based Testing
TXS >>  txsserver starting: "localhost" : 64180
TXS >>  Solver "z3" initialized : Z3 [4.8.7]
TXS >>  TxsCore initialized
TXS >>  LPEOps version 2019.07.05.02
TXS >>  input files parsed:
TXS >>  ["benchmarknew.txs"]
TXS >>  stepper MaxConcurrencyStressTestWithDeadlocks
TXS >>  Stepper started
TXS >>  step 20
TXS >>  .....1: Act { { ( Channel1, [] ) } }
TXS >>  .....2: Act { { ( Channel2, [] ) } }
TXS >>  .....3: Act { { ( Channel3, [] ) } }
TXS >>  .....4: Act { { ( Channel2, [] ) } }
TXS >>  no state or deadlock
TXS >>  FAIL: No Output (Quiescence)
```

Figure 3.3: The Output For The Maximum Concurrency Stress Test

and resource-related issues. However, due to the lack of randomized timing and order variations, some race conditions and other timing-sensitive faults may go undetected. Additionally, the absence of guarded actions and conditional logic in the current version slightly limits the ability to introduce interaction variability, which may impact test coverage to some extent.

3.1.3 Test Outcome

The output for the Maximum Concurrency Stress Test is shown in Figure 3.3. The 'MaxConcurrencyStressTest' model is designed to stress the system with maximum concurrency by utilizing all available channels (14 in total), running multiple sequences and synchronized processes in parallel, and handling complex data structures and operations.

We also argue that the number of channels and processes is adequate for the most common uses. The test utilizes 14 channels, which is a significant number to simulate a high-concurrency environment. These channels cover a variety of data types and operations, providing a comprehensive stress test. The processes include various models like parallel sequences, synchronized processes, and handling of complex data structures. The combination of these processes simulates a real-world scenario with high concurrency demands. We therefore argue that adding more channels and processes is not necessary unless one expects the system to handle even higher levels of concurrency in real-world scenarios. The current

```
TXS >>  TorXakis :: Model-Based Testing
TXS >>  txsserver starting: "localhost" : 64180
TXS >>  Solver "z3" initialized : Z3 [4.8.7]
TXS >>  TxsCore initialized
TXS >>  LPEOps version 2019.07.05.02
TXS >>  input files parsed:
TXS >>  ["benchmarknew.txs"]
TXS >>  stepper MaxConcurrencyStressTestWithDeadlocks
TXS >>  Stepper started
TXS >>  step 20
TXS >>  .....1: Act { { ( Channel1, [] ) } }
TXS >>  .....2: Act { { ( Channel2, [] ) } }
TXS >>  .....3: Act { { ( Channel3, [] ) } }
TXS >>  .....4: Act { { ( Channel2, [] ) } }
TXS >>  no state or deadlock
TXS >>  FAIL: No Output (Quiescence)
```

Figure 3.4: The Output For The Maximum Concurrency Stress Test With Deadlocks

setup should be sufficient for most high-concurrency environments. However, if even higher loads are anticipated then adding more channels to simulate more communication paths, introducing more complex synchronization patterns, and increasing the depth of nested processes should all be considered.

The output for the maximum concurrency stress test with deadlocks is shown in Figure 3.4. This result shows the system successfully handled the test cases. It shows that our concurrency test can indeed detect deadlock conditions. The deadlocks we are experiencing are by design, but they still confirm that TorXakis is identifying situations where processes cannot proceed due to circular dependencies. Without specific observations of inconsistent results or indefinitely waiting processes in our concurrency test, it is challenging to conclude if race conditions or starvation exist. These issues are more subtle and may require multiple runs with slightly different configurations to expose.

Just because a concurrency test does not detect deadlocks, race conditions, or starvation in a single test run, it doesn't guarantee that these issues don't exist. Some faults, particularly race conditions, are non-deterministic and may only occur under specific execution orders or timing conditions, making them difficult to detect in a single test. Concurrency issues are notoriously challenging to detect exhaustively, as the state space grows exponentially with the number of concurrent processes and channels. Even comprehensive tests might not cover all possible states or interactions.

This being said, this test is useful for the following reasons:

1. Early Detection of Issues: Even if it doesn't prove the absence of faults, a concurrency test is valuable for identifying obvious concurrency-related issues. For example, detecting deadlocks or performance bottlenecks under certain configurations can guide improvements.

2. Benchmarking System Limits: By pushing the system with high levels of concurrency, we can benchmark its performance and stability. This helps understand the boundaries within which the system can operate reliably, which is useful for real-world applications.

3. Identifying Bottlenecks: Running a concurrency test allows us to observe how resource usage and process interactions behave under load, which can help in identifying bottlenecks or areas where performance degrades.

4. Confidence in System Stability: Regularly passing concurrency tests without errors can provide a degree of confidence in the stability of the system under specific conditions, even if it doesn't prove fault-free operation under all scenarios.

3.2 Fault Tolerance Test

The fault tolerance test aims to verify the system's ability to handle faults by using backup channels and making choices between multiple channels. This kind of testing ensures that the system can still function correctly even if some components fail. The fault tolerance test models a system designed to continue functioning even in the presence of communication faults. This is achieved by introducing redundancy through the use of backup channels and simulating nondeterministic decision-making. The model comprises two main components: redundant communication channels and a nondeterministic selection mechanism. Each process in the redundant channel component attempts communication over a primary channel and, if necessary, falls back to a designated backup channel:

$$F_i = (p_i! \rightarrow F_i) \,\Box\, (b_i! \rightarrow F_i)$$

where p_i denotes the primary channel, b_i denotes the backup channel, and $\Box$ represents an external nondeterministic choice. This captures a fault-tolerant structure where either communication path can be used interchangeably to ensure progress.

In addition to redundancy, the model introduces a faulty choice process that selects nondeterministically between two alternative channels:

$$C = (c_1! \rightarrow C) \,\Box\, (c_2! \rightarrow C)$$

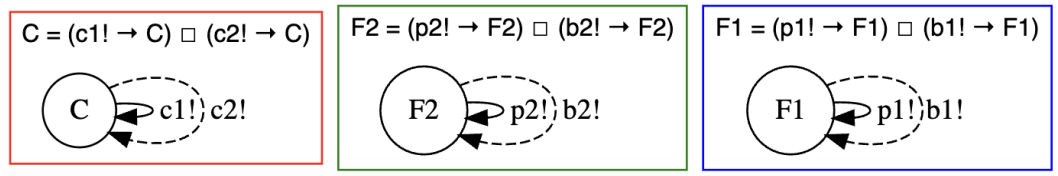

Figure 3.5: Local LTS Diagram for the Fault Tolerance Model

This process represents unpredictable or unstable communication conditions, simulating fault injection or dynamic channel failure. Channels c_1 and c_2 may fail independently, and the system must choose either path without predetermined logic.The complete fault tolerance model is defined as follows:

$$FaultToleranceModel = (F_1 \parallel F_2 \parallel C)$$

with the following definitions:

$$F_1 = (PrimaryChannel1! \rightarrow F_1) \Box (BackupChannel1! \rightarrow F_1)$$

$$F_2 = (PrimaryChannel2! \rightarrow F_2) \Box (BackupChannel2! \rightarrow F_2)$$

$$C = (Channel1! \rightarrow C) \Box (Channel2! \rightarrow C)$$

All three components operate in parallel and simulate redundancy and nondeterministic fault handling under concurrent execution. The structure ensures that failure of a single channel does not prevent the system from continuing execution, as long as alternative communication paths remain available. This configuration allows the system to simulate independent subsystems with redundancy and fault recovery. Each subsystem uses its own pair of primary and backup channels, and the faulty decision mechanism operates independently as well. The complete TorXakis code for this test is provided in Appendix A.3.

The corresponding LTS semantics of individual processes is shown in Figure 3.5. All channels are modeled as unidirectional (output-only), allowing isolated subsystem behavior and enabling parallelism without interference between the fault domains. The design ensures the model simulates realistic failover conditions. Figure 3.6 illustrates the global LTS diagram for the Fault Tolerance Model.

The behavior of this model is governed by several formal assumptions: First, each faulty sequence assumes that either the primary or backup channel can be used at any time, reflecting realistic hardware or communication fault conditions. Second, the system must remain operational by transitioning to the backup channel if the primary fails, implementing fault masking. Third, fault handling is performed internally by the model, requiring no external resolution. Fourth, the nondeterministic nature of the faulty choice component captures environmental uncertainty in channel availability. Fifth, the fault domains operate independently in parallel,

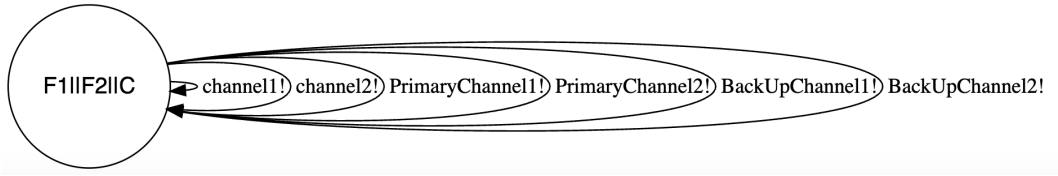

Figure 3.6: Global LTS Diagram for the Fault Tolerance Model

allowing for localized or system-wide fault simulation. Sixth, due to its recursive structure, the model is guaranteed not to deadlock under normal fault scenarios.

In conclusion, this model evaluates whether TorXakis can reliably simulate and recover from communication faults. It tests for graceful degradation in system performance and verifies the tool's ability to continue operation through alternative communication paths, aligning with ioco principles of observable behavior and system conformance.

3.2.1 Test Outcomes

The fault tolerance test code produced the output in Figure 3.7. The fault tolerance test verifies that the system can handle different types of faults: First, we ensure that the system can switch to a backup channel in case of a primary channel failure. Secondly, we detect nondeterministic faults, ensuring that the system can handle decisions where any channel might fail. Finally, we ensure that the system can handle complex scenarios with multiple failing components.

3.3 Scalability Test

The primary objective of the Scalability Test is to evaluate how well TorXakis performs under increasing system loads by incrementally scaling the number of channels and processes. This test aims to determine the system's practical limits for handling concurrency while maintaining stable performance, identifying potential bottlenecks, and providing insights into scalability improvements.

The test also seeks to answer key questions, such as:

- What is the maximum number of channels and processes TorXakis can effectively handle under the current hardware and configuration?
- Are the observed limits inherent to TorXakis, or are they dependent on the computational resources available?
- What specific adjustments or tuning can improve performance under high loads?

```
TXS >>  TorXakis :: Model-Based Testing
TXS >>  txsserver starting: "localhost" : 55152
TXS >>  Solver "z3" initialized : Z3 [4.8.7]
TXS >>  TxsCore initialized
TXS >>  LPEOps version 2019.07.05.02
TXS >>  input files parsed:
TXS >>  ["benchmarknew.txs"]
TXS >>  stepper SpecCombinedFaultTolerance
TXS >>  Stepper started
TXS >>  step 10
TXS >>  .....1: Act { { ( BackupChannel2, [] ) } }
TXS >>  .....2: Act { { ( PrimaryChannel1, [] ) } }
TXS >>  .....3: Act { { ( BackupChannel1, [] ) } }
TXS >>  .....4: Act { { ( Channel1, [] ) } }
TXS >>  .....5: Act { { ( Channel1, [] ) } }
TXS >>  .....6: Act { { ( BackupChannel2, [] ) } }
TXS >>  .....7: Act { { ( Channel1, [] ) } }
TXS >>  .....8: Act { { ( PrimaryChannel2, [] ) } }
TXS >>  .....9: Act { { ( PrimaryChannel1, [] ) } }
TXS >>  ....10: Act { { ( Channel1, [] ) } }
TXS >>
TXS >>  PASS
```

Figure 3.7: The Output For The Combined Fault Tolerance Test

To conduct the scalability test, a model was implemented using a combination of sequential, parallel, and synchronized processes defined over multiple channels. We use sequences, nondeterministic choices, parallel processes, synchronized blocks, and modular components, each representing increasing complexity and resource demand. Formaly, let us define the basic processes as follows:

- Sequence processes:

$$S_i = ch_i! \rightarrow S_i \quad \text{for } i = 1, 2, 3$$

- Choice processes:

$$C_{ij} = (ch_i! \rightarrow C_{ij}) \Box (ch_j! \rightarrow C_{ij}) \quad \text{for } (i, j) = (4, 5), (6, 7), (8, 9)$$

The parallel and synchronized constructs are defined as follows:

- Parallel processes:

$$P_{10} = ch_{10}! \rightarrow P_{10}, \quad P_{11} = ch_{11}! \rightarrow P_{11}$$

- Alternating parallel blocks:

$$AP_{12,13} = (ch_{12}! \rightarrow ch_{13}! \rightarrow AP_{12,13})$$

$$AP_{14,15} = (ch_{14}! \rightarrow ch_{15}! \rightarrow AP_{14,15})$$

- Synchronized processes:

$$Sync_{16} = ch_{16}! \rightarrow Sync_{16}, \quad Sync_{17} = ch_{17}! \rightarrow Sync_{17}$$

We also define higher-level modules as follows:

- Enable-based: $E_{24,25}$
- Hidden/composite: $H_{26}, H_{27\text{–}30}, H_{31}$
- Synchronized sequence: $SS_{32\text{–}35}$
- Combined concurrency: CC (channels ch_{36}–ch_{44})
- Fault Tolerance block: FT (channels ch_{45}–ch_{50})

Finally, the overall system is modeled as:

$$\begin{aligned} ScalabilityModel = \\ & (S_1 \parallel S_2 \parallel S_3 \parallel C_{4,5} \parallel C_{6,7} \parallel C_{8,9} \parallel P_{10}^5 \parallel P_{11}^5 \\ & \parallel AP_{12,13}^5 \parallel AP_{14,15}^5 \parallel \cdots \parallel FT) \end{aligned}$$

The numerical superscript $(\cdot^n)$ indicates multiple concurrent instances, where each copy behaves independently but follows the same process definition. Refer to Appendix A.4 for the full TorXakis model.

The test explores the structural and behavioral boundaries of TorXakis by evaluating system performance under increasingly complex workloads. It employs 19 output channels to simulate sequences, nondeterministic branching, parallel and alternating execution, and synchronization. Channels are kept unidirectional (output only) and distinct to prevent interference, except when specifically reused under synchronization. To simulate realistic high-load behavior, channels are grouped based on function. Sequential processes operate on distinct channels to represent simple pipelines. Choice processes simulate environmental or internal decisions by randomly selecting between communication paths. Parallel constructs test fixed and alternating execution patterns across channels. Synchronization channels evaluate coordinated communication under concurrent stress. Constructs like `parallelN` and `synchronizedN` manage bounded parallelism while exploring increased concurrency. State-space explosion is implicitly tracked by monitoring system behavior as process count grows.

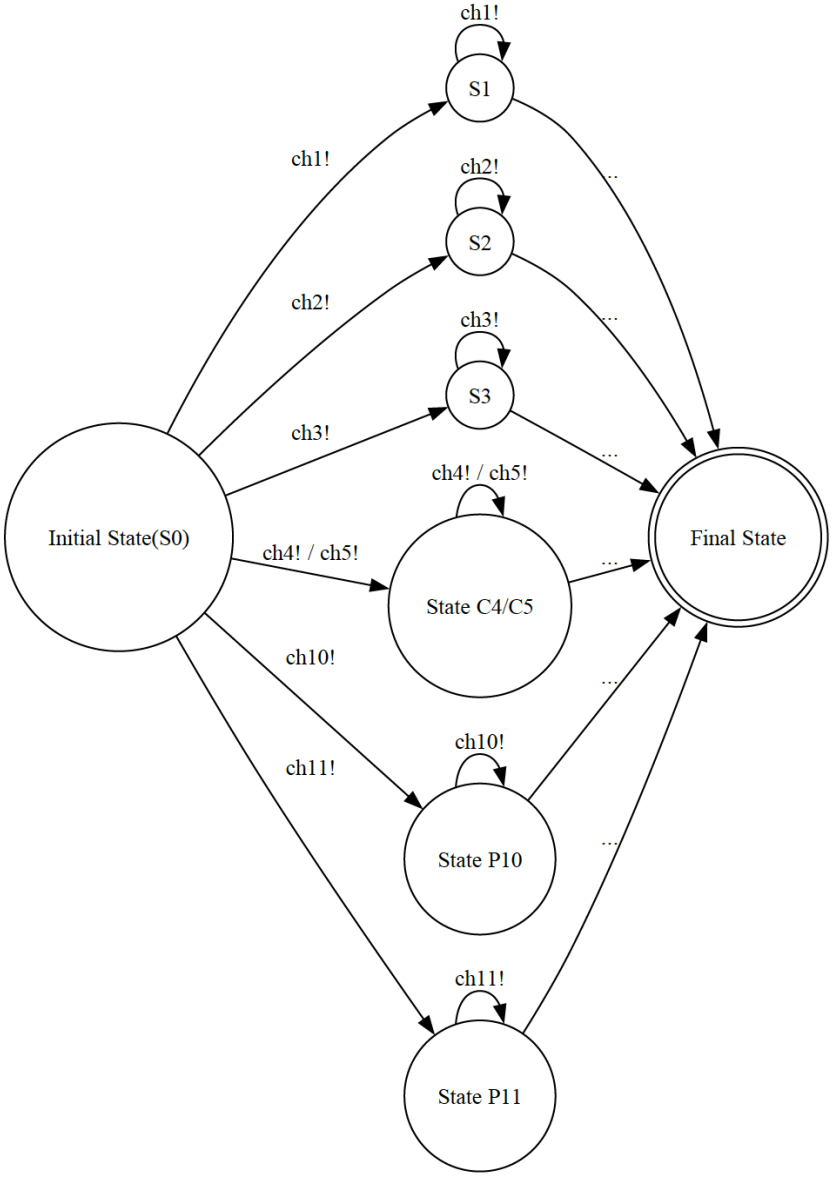

Figure 3.8: Simplified LTS Diagram for the Scalability Test

Alternating parallelism introduces branching complexity, while synchronized blocks evaluate the tool's ability to manage coordinated processes. Mixing deterministic and nondeterministic loads tests execution stability and modeling flexibility. The combined goal is to understand how TorXakis manages concurrency across diverse constructs and identify its resource limits. Figure 3.8 shows the simplified LTS for the Scalability Test.

Due to the algebraic model's considerable complexity and size, the LTS diagram presented here has been simplified to highlight the key structural behaviors rather than exhaustively representing all process interactions. This abstraction focuses on capturing the essential concurrency patterns, synchronization schemes, and process categories that define the scalability evaluation. Such simplification improves visual clarity and avoids overwhelming detail, while still communicating the model's architectural depth and testing scope.

Successful execution without crashes, deadlocks, or starvation—while maintaining observable outputs on defined channels—indicates that TorXakis handles the complexity gracefully. If the model reaches hardware or runtime constraints, it should either stall via quiescence or report resource-related errors. This behavior validates the framework's capacity for scalable testing and highlights architectural bottlenecks or runtime limits.

3.3.1 Test Outcomes

Based on our findings that TorXakis can handle up to 19 channels effectively, we can conclude that this is a practical limit for scalability testing within the given constraints. Here are some key points and conclusions regarding the scalability test with 19 channels and processes:

TorXakis has a practical limit regarding the number of channels and processes it can handle efficiently. Testing with 50 channels exceeds this limit, leading to performance issues or failures. The output of scalability test with 50 channels is shown in Figure 3.9. Testing with 19 channels ensures that the system remains within operational limits, providing a realistic and practical scenario for scalability testing. The output of scalability test with 19 channels is shown in Figure 3.10.

Even with 19 channels, the test can cover a wide range of scenarios including sequences, choices, parallel processes, and synchronization. This setup also can simulate a realistic load that the system might face in a production environment, ensuring that the performance metrics are meaningful.

We argue that the use of 19 channels ensures the system is not overloaded beyond its capacity, providing meaningful results without causing crashes or undefined behaviors, and at the same time offers comprehensive testing by covering various process combinations and interactions, and thus providing a thorough evaluation of the system's performance and scalability. This limit of 19 channels appears to be a practical limit within the specific constraints of our current setup and the version

```
TXS >>  TorXakis :: Model-Based Testing
TXS >>  txsserver starting: "localhost" : 55372
TXS >>  Solver "z3" initialized : Z3 [4.8.7]
TXS >>  TxsCore initialized
TXS >>  LPEOps version 2019.07.05.02
TXS >>  input files parsed:
TXS >>  ["benchmarknew.txs"]
TXS >>  stepper SpecCombinedScalabilityTest
```

Figure 3.9: The Output For The Combined Scalability Test

```
TXS >>  TorXakis :: Model-Based Testing
TXS >>  txsserver starting: "localhost" : 55462
TXS >>  Solver "z3" initialized : Z3 [4.8.7]
TXS >>  TxsCore initialized
TXS >>  LPEOps version 2019.07.05.02
TXS >>  input files parsed:
TXS >>  ["benchmarknew.txs"]
TXS >>  stepper SpecCombinedScalabilityTest
TXS >>  Stepper started
TXS >>  step 3
TXS >>  .....1: Act { { ( Channel14, [] ) } }
TXS >>  .....2: Act { { ( Channel10, [] ) } }
TXS >>  .....3: Act { { ( Channel14, [] ) } }
```

Figure 3.10: The Output Of Scalability Test With 19 Channels

of TorXakis we are using (the 2017 version). It's likely influenced by factors like the system's memory capacity, processing power, and how the TorXakis software itself is architected in that version. TorXakis doesn't specify a hard limit on the number of channels or processes it can handle.

TorXakis does not provide direct configuration options for stack size, memory allocation, or threading limits; however, its performance can be optimized through several approaches. System resources play a crucial role in TorXakis's efficiency. Ensuring sufficient RAM allows the tool to handle complex models more effectively, and upgrading memory can significantly improve its ability to process larger models. Additionally, processor performance is a key factor, as a faster CPU enhances the tool's execution speed, particularly for intricate models.

Model optimization also influences performance. Reducing model complexity

decreases computational load, leading to improved efficiency. A modular approach, in which large models are divided into smaller, more manageable components, enables TorXakis to process them more effectively, reducing potential performance bottlenecks.

TorXakis relies on SMT solvers such as Z3 and CVC4 for verification tasks. Properly configuring these solvers is essential to enhancing performance, particularly by ensuring support for SMTLIB version 2.5, Algebraic Data Types (ADTs), and String theory. While TorXakis does not offer explicit threading configurations, running multiple instances in parallel–if system resources permit–can help simulate concurrent processes and improve overall execution efficiency.

Another consideration is environment configuration at the operating system level. Adjusting OS resource allocation, such as increasing the maximum number of open files or processes, can be beneficial when working with numerous channels. If TorXakis is built from source, using an appropriate Haskell-based build tool like Stack is necessary. Stack is a build and dependency management tool for Haskell projects, ensuring compatibility with the correct compiler version and package dependencies.

One other bottleneck likely lies in the handling of synchronization and concurrency mechanisms. Channels requiring frequent synchronized steps or choices may introduce delays, especially when concurrency levels are high. This suggests that when approaching the limits revealed by the test one should consider redesigning these synchronized interactions or replacing some of them with less resource-intensive alternatives (e.g., reducing the number of synchronized steps or using asynchronous sequences) could alleviate strain. Additionally, where possible, avoiding "choice" structures that lead to nondeterministic decision-making can reduce complexity and resource usage.

3.4 Resource Utilization Test

The Resource Utilization Test was designed to evaluate the system's performance under resource-intensive scenarios by simulating sequences, parallel processes, and synchronized tasks across multiple channels. This test aims to analyze how the system manages CPU, memory, and other resources under heavy loads while identifying potential bottlenecks and ensuring performance stability. The test focuses on the following:

1. Evaluating Resource Usage: Monitoring CPU, memory, and disk utilization under various concurrency scenarios.

2. Identifying Bottlenecks: Detecting processes or operations that lead to resource exhaustion or performance degradation.

3. Ensuring Stability: Verifying the system's ability to handle complex operations without deadlocks or failures.

We define the test formally as follows: For $i \in \{1, \ldots, 9\}$, let

$$S_i = ch_i! \rightarrow S_i$$

which represents an infinite loop of simple outputs on channel ch_i. We then define the synchronized resource-intensive processes

$$R_i(n) = \begin{cases} STOP, & n = 0 \\ ch_i! \rightarrow R_i(n-1), & n > 0 \end{cases}$$

and then let $R_i = R_i(5)$ for $i \in \{1, \ldots, 9\}$. Parallel data-handling processes are defined for each $j \in \{1, 2, 3\}$ as follows:

$$P_j(m) = \begin{cases} STOP, & m = 0 \\ chInt_j! \rightarrow P_j(m-1), & m > 0 \end{cases}$$

and then let $P_j = P_j(10)$. We also define two complex data sequence processes:

$$C_1 = ch_{10Ints}! \rightarrow ch_{10Ints}! \rightarrow \cdots \rightarrow STOP$$

$$C_2 = ch_{10Ints\_b}! \rightarrow ch_{10Ints\_b}! \rightarrow \cdots \rightarrow STOP$$

Finally, we define the following nested synchronization sequences:

$$N_i = ch_i! \rightarrow N_i$$

for $i \in \{1, \ldots, 9\}$.

All the processed defined above run in parallel and synchronize with each other as follows:

$$\begin{aligned} ResourceUtilizationModel = {} & (S_1 \parallel S_2 \parallel \cdots \parallel S_9) \\ & \|_{\{ch_1,\ldots,ch_9\}} (R_1 \parallel R_2 \parallel \cdots \parallel R_9) \\ & \|_{\{chInt_1,chInt_2,chInt_3\}} (P_1 \parallel P_2 \parallel P_3) \\ & \|_{\{ch_{10Ints},ch_{10Ints\_b}\}} (C_1 \parallel C_2) \\ & \|_{\{ch_1,\ldots,ch_9,chInt_1,\ldots,ch_{10Ints\_b}\}} (N_1 \parallel \cdots \parallel N_9) \end{aligned}$$

where $\|_X$ denotes parallel composition synchronized over the set of channels X.

The model launches nine sequence processes, nine synchronized "heavy" processes, three parallel data-processing tasks, two complex structured-data sequences, and nine nested synchronized processes simultaneously. The number of channels and processes is more or less arbitrary and aims to provide a sufficiently complex

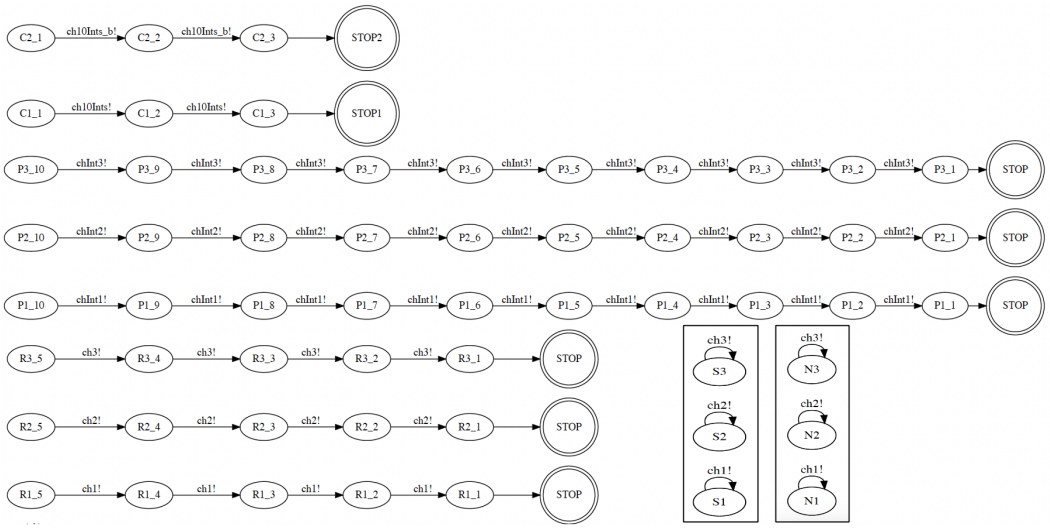

Figure 3.11: Local LTS of the Resource Utilization Test

test without making the model unwieldy. The complete TorXakis code for this test is provided in Appendix A.5.

To further illustrate the structure and behavior of the Resource Utilization Test, we provide two LTS diagrams. The first diagram (Figure 3.11) presents the *local LTS*, which models the execution logic of a single process under resource constraints, highlighting local sequencing, synchronization, and channel communication. The second diagram (Figure 3.12) depicts the *global LTS*, which models the overall system behavior when multiple resource-intensive processes execute concurrently and interact through shared channels. This global view exposes potential points of contention, synchronization bottlenecks, and parallel workload dynamics under high-load conditions.

The Resource Utilization Test is designed to simulate sequential, synchronized, parallel, and data-heavy operations across fifteen output-only channels. These channels are categorized into three types: basic sequential channels (Channel1 through Channel9), which are employed by independent sequence and synchronized processes to emulate task-level operations; integer data channels (ChannelInt1, ChannelInt2, and ChannelInt3), which handle parallel numeric data streams to test memory and computation throughput; and complex structured channels (Channel10Ints and Channel10Ints_b), which support composite or nested data formats such as arrays or records. All channels are reused throughout various process layers, including sequence execution, nested tasks, and synchronization stages. This reuse increases contention and mimics a high-load system environment with realistic constraints. The model operates within a closed system, with no

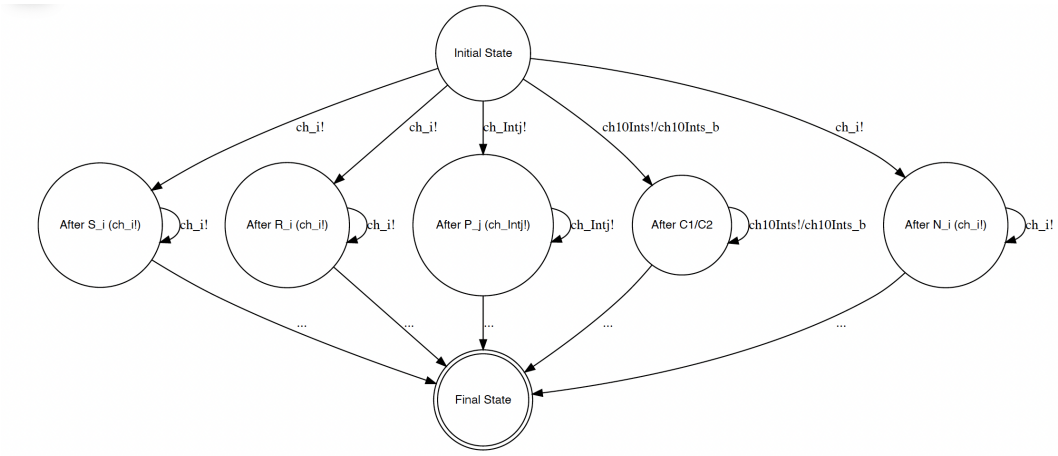

Figure 3.12: Global LTS of the Resource Utilization Test

external feedback or inputs, ensuring that all data generation and synchronization occur internally. While the test does not explicitly measure CPU or memory usage, system performance is inferred through observable metrics such as CPU, RAM, and execution time.

3.4.1 Test Outcomes

We executed the test in stages. For an initial load testing, we started with a small number of channels and gradually increased to test scalability. We ended up in high load scenarios, where we scaled up to maximum channels and processes to push resource limits. System metrics were tracked using macOS Activity Monitor, including CPU, memory, and disk usage, to analyze performance and identify bottlenecks. This test configuration enables one to evaluate how the system scales as the number of processes and channels increases. It's a meaningful size to show whether the system handles increasing loads without breaking or slowing down. Testing simple and complex data structures separately gives you insights into how the system performs when handling more computationally intensive tasks.

The output of the resource utilization test is shown in the Figure 3.13. To monitor the resource utilization, we typically run these models in our TorXakis environment and use system monitoring tools to observe CPU, memory, and other resource usage. Tools like 'top', 'htop', 'vmstat', or system-specific monitoring tools (e.g., Task Manager on Windows, Activity Monitor on macOS) can be used to track the resource consumption.

As the number of active channels and concurrent processes increased, particularly beyond 19 channels, the behavior of the model became increasingly unstable. In multiple test runs, certain processes failed to execute as expected or halted

```
TXS >>  TorXakis :: Model-Based Testing
TXS >>  txsserver starting: "localhost" : 55690
TXS >>  Solver "z3" initialized : Z3 [4.8.7]
TXS >>  TxsCore initialized
TXS >>  LPEOps version 2019.07.05.02
TXS >>  input files parsed:
TXS >>  ["benchmarknew.txs"]
TXS >>  stepper ResourceUtilizationTest
TXS >>  Stepper started
TXS >>  step 10
TXS >>  .....1: Act { { ( Channel5, [] ) } }
TXS >>  .....2: Act { { ( Channel6, [] ) } }
TXS >>  .....3: Act { { ( Channel4, [] ) } }
TXS >>  .....4: Act { { ( Channel1, [] ) } }
TXS >>  .....5: Act { { ( Channel6, [] ) } }
TXS >>  .....6: Act { { ( Channel2, [] ) } }
TXS >>  .....7: Act { { ( Channel7, [] ) } }
TXS >>  .....8: Act { { ( Channel8, [] ) } }
TXS >>  .....9: Act { { ( Channel7, [] ) } }
TXS >>  ....10: Act { { ( Channel5, [] ) } }
TXS >>
TXS >>  PASS
```

Figure 3.13: The Output Of The Resource Utilization Test

midway without completing their defined sequences. In extreme cases, the system entered a STUCK state, where no further transitions were possible, effectively freezing the simulation. This behavior indicates that the underlying model is not able to sustain proper execution paths once concurrency surpasses a certain threshold, likely due to resource exhaustion or process contention.

From a qualitative perspective, several structural weaknesses were identified within the model. One of the most notable issues was the instability introduced by nested parallelism combined with shared channel usage. When several parallel processes attempted to access or synchronize over the same set of channels, the simulation exhibited unpredictable outcomes, ranging from delays in communication to complete deadlocks. This suggests that the current design struggles to manage high concurrency when shared communication resources are involved. Additionally, synchronization bottlenecks were observed when too many concurrent tasks were activated simultaneously. These bottlenecks resulted in significant slowdowns and, in some cases, process starvation, where certain tasks could no longer proceed

Number of Channels	CPU Usage (%)	Memory Usage (MB)
1	0.9	11
3	0.9	11.1
5	1	10.2
7	1.5	10.6
9	1.8	11
11	377	337.3
13	383.6	355.1
15	358.6	750.3
17	383	2830
19	392.1	7010

Table 3.2: CPU and Memory Usage by Number of Channels

due to the unavailability of their required synchronization partners.

Together, these insights highlight the fragility of the system under high concurrency scenarios and provide a strong case for either redesigning the process structure to reduce shared dependencies or enhancing the tool.

As shown in Table 3.2, Figure 3.14, and Figure 3.15 both CPU and memory usage gradually increase as the number of channels rises during the combined scalability and resource utilization test. While CPU usage remains relatively low for smaller models (1 to 9 channels), a noticeable increase starts beyond 9 channels, reaching 377% at 11 channels. It indicates that approximately 3.7 processor cores were simultaneously active. This high utilization confirms the heavy computational demand of the model, particularly due to its parallel and synchronized processes. Such results reflect the system's ability to exploit multi-core environments efficiently.

As shown in Table 3.2 this trend becomes significantly more pronounced when the model is scaled to 11 or more channels. The increase in memory requirements appears exponential. Probably the same holds for CPU utilization, but that does not show because of CPU saturation. While memory usage initially increases in small, linear steps between 10.2 MB and 11.1 MB, the rise becomes exponential once the system approaches its concurrency limit. At 11 channels, memory usage not only increases in raw allocation but also triggers secondary memory management mechanisms such as memory compression and swap usage. These mechanisms indicate that the physical RAM is no longer sufficient to accommodate the active and inactive memory blocks required by the model, especially when nested parallel and synchronized processes are active. This suggests that the TorXakis execution engine faces a combinatorial explosion of state-space management and inter-process synchronization tracking. This growth pattern, combined with CPU saturation ultimately leads to failure to execute simulations beyond 19 channels caused by hardware and runtime limitations.

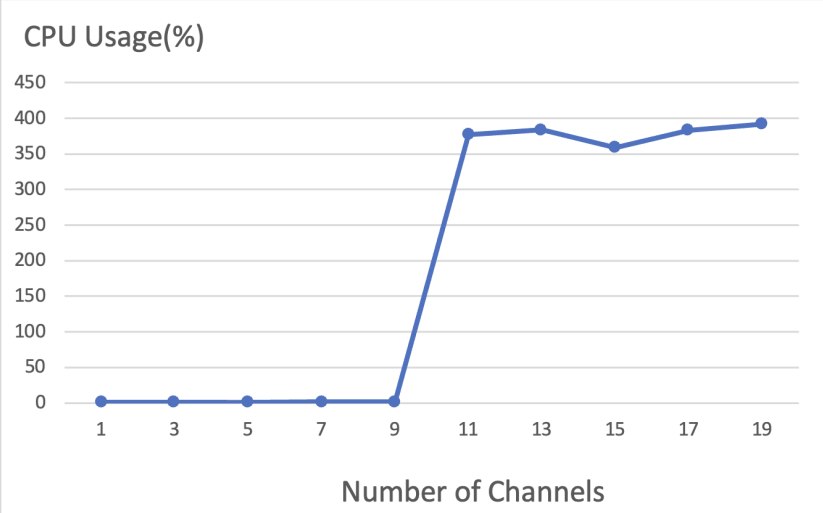

Figure 3.14: CPU Usage vs. Number of Channels

3.5 Model Verification Test

The Model Verification Test aims to ensure the correctness, reliability, and robustness of the system by verifying critical properties and behaviors within the model. This test evaluates the following key aspects:

1. Liveness: Ensures that processes continue to make progress and do not get stuck in an idle state.
2. Deadlock Freedom: Verifies that the system does not encounter situations where progress halts due to circular dependencies.
3. Correct Synchronization: Confirms that processes synchronize accurately on shared channels.
4. Data Integrity: Validates that data transmitted through channels complies with specified integrity constraints.
5. Choice Handling: Ensures that the system correctly handles decision-making and alternative actions.
6. Concurrency: Tests whether concurrent processes operate without interference.
7. Extended and Nested Synchronization: Validates synchronization among multiple or nested processes for complex workflows.

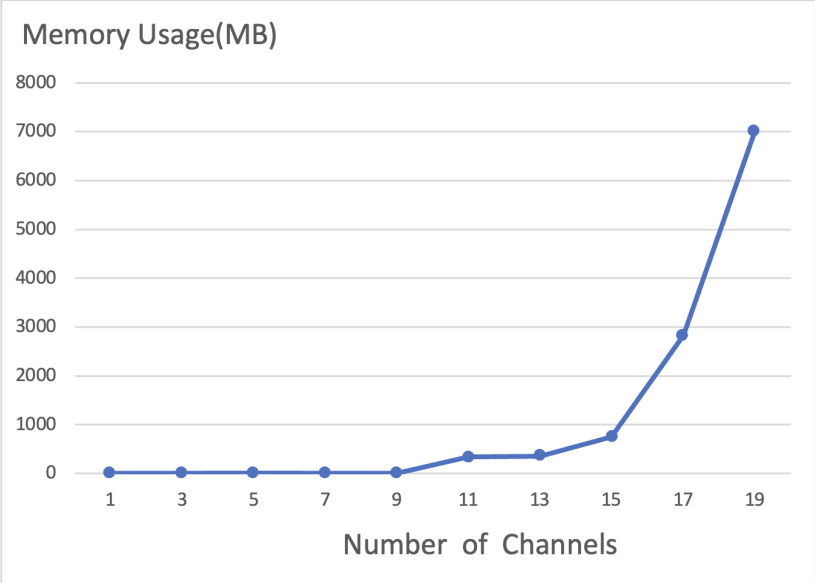

Figure 3.15: Memory Usage vs. Number of Channels

By expanding the scope of verification, this test ensures that critical functional and behavioral aspects of the system are thoroughly assessed, providing confidence in its correctness and ability to handle complex scenarios.

To formally verify the behavioral properties of the system, eight distinct verification scenarios were modeled as independent processes and then composed in parallel to form a comprehensive verification suite. Each scenario targets a specific property, and together they cover both correctness and robustness under various conditions.

The liveness property ensures that the process interacting through channel ch_1 never stalls, expressed by the recursive process $L = ch_1! \rightarrow L$. Deadlock freedom is verified using two independent processes that terminate after a single action, avoiding circular waiting, as defined by $ND = (ch_1! \rightarrow STOP) \parallel (ch_2! \rightarrow STOP)$.

Pairwise synchronization is tested by ensuring that two processes synchronize correctly over channels ch_1 and ch_2, modeled as $SY = (ch_1! \rightarrow ch_2! \rightarrow SY) \parallel_{[ch_1, ch_2]} (ch_2! \rightarrow ch_1! \rightarrow SY)$. Data integrity is validated through channel input ch_{Int1}, where each received integer respects the defined constraints, expressed as $DI = ch_{\text{Int1}}?x \rightarrow DI$.

To test non-deterministic behavior, a choice-handling scenario is implemented where the process can choose between two sending actions, modeled as $CH = (ch_1! \rightarrow CH) \Box (ch_2! \rightarrow CH)$. Concurrency is evaluated by verifying that two processes can execute in parallel without interference, modeled by $CO = (ch_1! \rightarrow STOP) \parallel (ch_2! \rightarrow STOP)$.

In more complex verification, multi-process synchronization is addressed by a process that sequentially synchronizes over three channels, represented by $MS = (ch_1! \rightarrow ch_2! \rightarrow ch_3! \rightarrow STOP) \parallel_{[ch_1,ch_2,ch_3]} (ch_2! \rightarrow ch_3! \rightarrow ch_1! \rightarrow STOP)$. Nested synchronization behavior is tested using embedded synchronization scopes, as seen in $NS = (ch_1! \rightarrow STOP) \parallel_{[ch_1]} (ch_2! \rightarrow STOP)$, ensuring proper nesting behavior.

To verify the system's ability to detect deadlocks, an intentional deadlock model is constructed based on circular synchronization dependencies. The test introduces two processes that attempt to synchronize over two shared channels but in reversed order, thereby forming a classic circular wait pattern. Formally, the model is defined as:

$$ID = (ch_1! \rightarrow ch_2! \rightarrow STOP) \parallel_{\{ch_1,ch_2\}} (ch_2! \rightarrow ch_1! \rightarrow STOP)$$

In this configuration, the first process attempts to synchronize on channel ch_1, followed by ch_2, while the second process initiates communication on ch_2 and then proceeds to ch_1. Due to the synchronization set $\{ch_1, ch_2\}$, both communications require cooperation between the processes. However, because each process is waiting for the other to synchronize on a channel that it has not yet reached, a circular dependency is created. As a result, neither process can proceed, and the system enters a deadlocked state. This setup effectively simulates one of the most common deadlock scenarios in concurrent systems and allows us to evaluate whether the system under test can detect and report the absence of further progress as a deadlock.

The complete verification suite combines all these individual processes into a parallel composition:

$$VerifyAll = L \parallel ND \parallel SY \parallel DI \parallel CH \parallel CO \parallel MS \parallel NS \parallel ID$$

The Model Verification Test utilizes a variety of output channels to verify essential behavioral properties of reactive systems. These channels are designed to assess distinct aspects of system behavior under controlled conditions. Channel1 plays a central role in validating liveness and sequencing by ensuring the system can make continual progress without stalling. Channels such as Channel2 and Channel3 are employed in deadlock and concurrency-related models to test for inter-process blocking, synchronization failures, and race conditions. ChannelInt1 is used specifically to validate data integrity by transmitting typed integer inputs, ensuring they conform to the expected constraints. Additionally, Channel1 through Channel3 are reused in multi-process synchronization scenarios to verify coordination across multiple actors and shared communication resources. Figures 3.16 and 3.17 illustrate these concepts in detail.

All channels are output-oriented and reused across multiple test models. Their reuse is intentional, enabling the system to be tested under varied combinations of interactions to uncover hidden issues. This controlled reuse ensures that each

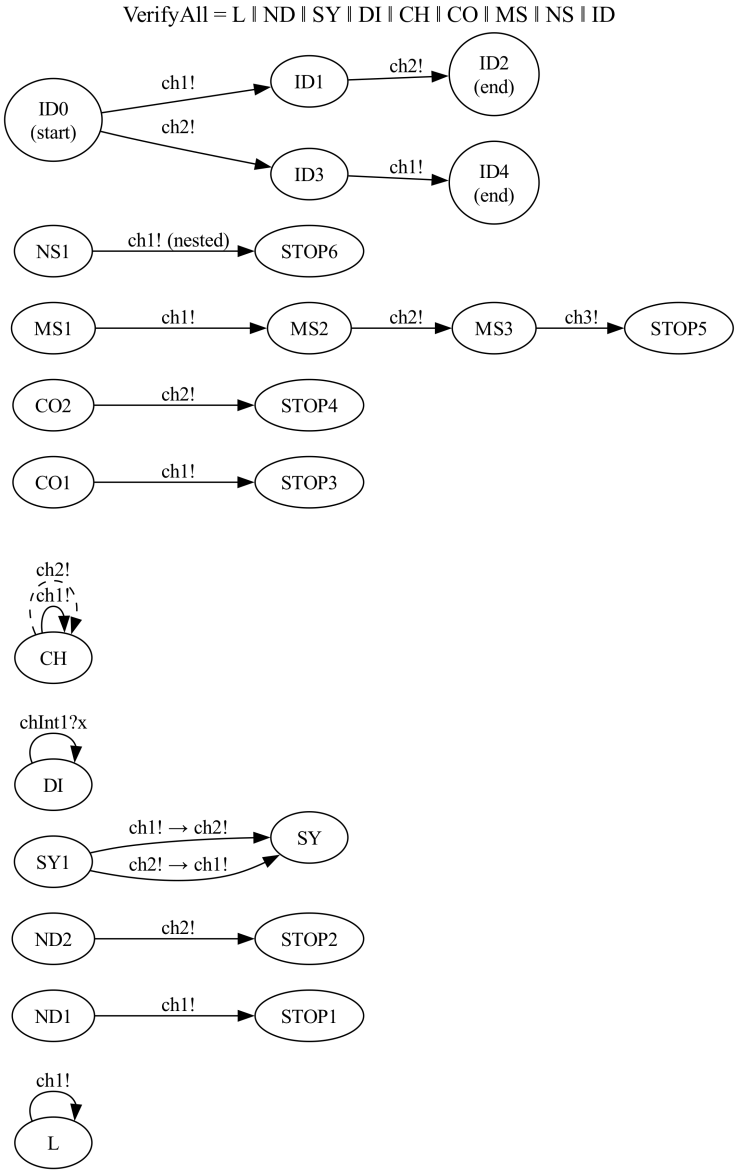

Figure 3.16: Local LTS of the Model Verification Test

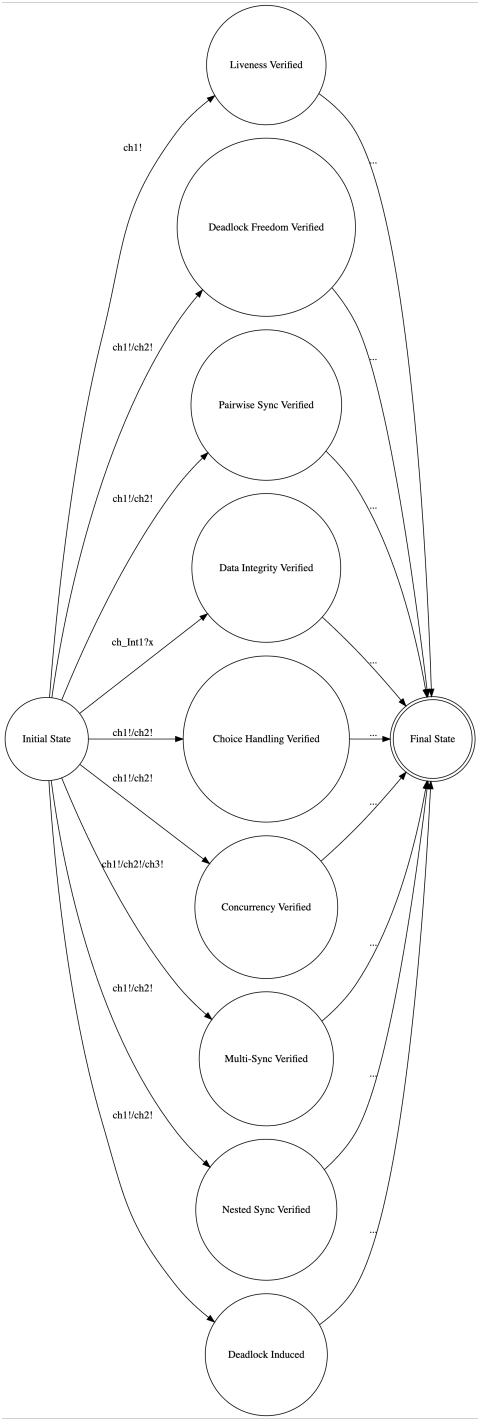

Figure 3.17: Global LTS of the Model Verification Test

property is verified in isolation while still accounting for potential conflicts during shared execution.

Each verification subtest is grounded in a specific formal assumption that corresponds to the property being tested. For liveness verification, the model assumes that repeated activation of a channel (e.g., `Channel1`) implies indefinite progress, modeling a system that never halts unexpectedly. The deadlock-freedom verification ensures that models such as `verifyNoDeadlock` and `induceDeadlock` either terminate normally or demonstrate intentional blocking without circular dependencies. Synchronization semantics are tested by ensuring the correct ordering and shared commitment to actions across channels, particularly in pairwise, multiprocess, and nested synchronization scenarios.

The verification of nondeterministic branching assumes that environmental uncertainty is properly modeled. For instance, the choice-handling process uses the `##` operator to explore all possible paths in a behaviorally sound way. Data validity is tested by confirming that received inputs comply with range, type, or format constraints, even under random input values.

Concurrency and race condition testing focuses on simulating parallel execution across processes. It verifies that simultaneous actions on different channels do not lead to interference or unintended interactions. Reuse of shared channels like `Channel1` is handled with caution: each submodel is executed in isolation to maintain behavioral determinism and ensure that property-specific validation is not compromised. Moreover, each test is designed to assess a single formal property, supporting modular test execution, easier debugging, and accurate fault localization.

The complete TorXakis code for this test is provided in Appendix A.6.

3.5.1 Test Outcomes

The output of the Model Verification Test is shown in Figure 3.18 . The output of the Deadlock Inducing Test is shown in Figure 3.19.

The test we provided involves multiple channels and processes, covering a wide range of system behaviors. We argue that this setup is sufficient for model verification, as follows: The test uses a sufficient variety of channels (‘Channel1‘, ‘Channel2‘, ‘ChannelInt1‘, ‘ChannelInt2‘, etc.) to verify synchronization, choice handling, and data integrity. Each channel is designed to test different aspects of the system, from transmitting data to handling multiple inputs. Having this variety allows you to test various forms of communication, synchronization, and process interaction. The test involves multiple processes interacting in different ways (sequential, parallel, synchronized, etc.). This covers a wide range of possible system states and ensures that the model can handle the complexities of real-world systems, where processes often run concurrently and need to synchronize with each other. Finally, the key behaviors of synchronization, data flow, parallelism, and

```
TXS >>  TorXakis :: Model-Based Testing
TXS >>  txsserver starting: "localhost" : 54191
TXS >>  Solver "z3" initialized : Z3 [4.8.7]
TXS >>  TxsCore initialized
TXS >>  LPEOps version 2019.07.05.02
TXS >>  input files parsed:
TXS >>  ["benchmarknew.txs"]
TXS >>  stepper SpecVerifyDataIntegrity
TXS >>  Stepper started
TXS >>  step 5
TXS >>  .....1: Act { { ( ChannelInt1, [ -29 ] ) } }
TXS >>  .....2: Act { { ( ChannelInt1, [ 77 ] ) } }
TXS >>  .....3: Act { { ( ChannelInt1, [ 17 ] ) } }
TXS >>  .....4: Act { { ( ChannelInt1, [ -27 ] ) } }
TXS >>  .....5: Act { { ( ChannelInt1, [ 67 ] ) } }
TXS >>
TXS >>  PASS
```

Figure 3.18: The Output Of The Model Verification Test

choice handling are all covered by the current number of processes and channels. These are critical aspects of most distributed or concurrent systems, and testing these core behaviors ensures the system is robust.

The model verification test demonstrates that the system progresses as expected without stalling (liveness). No deadlocks are present, and all processes can proceed without being blocked by waiting on each other (deadlock freedom). Processes synchronize correctly, meaning that channels with shared behavior properly wait for and react to each other's events (synchronization). Only valid data is transmitted and processed by the system, ensuring data integrity. The system can handle multiple possible inputs and make choices based on those inputs (choice handling). Multiple processes can run in parallel without interfering with each other (concurrency and parallelism). Complex synchronization involving nested or multiple processes behaves as expected (extended synchronization).

3.6 Integration Test

The Integration Test aims to evaluate how different components of the system interact with one another to ensure correct and predictable behavior. This includes verifying the system's ability to handle key integration challenges such as deadlocks, race conditions, incorrect message passing, and branching errors. The test

```
TXS >>  TorXakis :: Model-Based Testing
TXS >>  txsserver starting: "localhost" : 52176
TXS >>  Solver "z3" initialized : Z3 [4.8.7]
TXS >>  TxsCore initialized
TXS >>  LPEOps version 2019.07.05.02
TXS >>  input files parsed:
TXS >>  ["benchmarknew.txs"]
TXS >>  stepper SpecInduceDeadlock
TXS >>  Stepper started
TXS >>  step 10
TXS >>  .....1: Act { { ( Channel2, [] ) } }
TXS >>  .....2: Act { { ( Channel1, [] ) } }
TXS >>  .....3: Act { { ( Channel1, [] ) } }
TXS >>  no state or deadlock
TXS >>  FAIL: No Output (Quiescence)
```

Figure 3.19: The Output Of The Deadlock Inducing Test

focuses on identifying faults that emerge from the interaction of multiple processes, channels, and constructs, ensuring system robustness and correctness in integrated scenarios. Specific objectives include:

1. Deadlock detection: To ensure the system does not get stuck due to unresolvable synchronization or waiting issues.
2. Race condition identification: To detect non-deterministic behaviors resulting from parallel processes interacting without proper synchronization.
3. Incorrect message passing: To verify that processes handle mismatched or unexpected data types correctly and report errors when violations occur.
4. Branching errors: To confirm that decision-making processes (e.g., choice operations) transition to appropriate branches based on input values and reject invalid inputs.

Formally, we first define the core component processes involved in this model. The first process, a simple sequence over Channel1, is given by

$$Seq_1 = ch_1! \rightarrow Seq_1$$

The second component introduces nondeterminism via a choice construct operating over Channel2 and Channel3:

$$Choice_{2,3} = (ch_2! \rightarrow Choice_{2,3}) \,\square\, (ch_3! \rightarrow Choice_{2,3})$$

A third component models a synchronized three-step sequence, also over Channel1:

$$Sync_1 = ch_1! \rightarrow ch_1! \rightarrow ch_1! \rightarrow Sync_1$$

In addition, we introduce a parallel process executed twice over Channel4, which models concurrent subsystem behavior:

$$Par_4 = ch_4! \rightarrow ch_4! \rightarrow Par_4$$

These processes are composed using both interleaving and synchronized parallel composition. The complete model is therefore defined as follows:

$$IntegrationDeadlock = \big((Seq_1 \parallel Choice_{2,3} \parallel Sync_1) \parallel_{\{ch_1,ch_2,ch_3\}} Seq_1\big) \parallel_{\{ch_1,ch_4,ch_5\}} Par_4$$

Recall that $\parallel_X$ denotes a parallel composition synchronized over the set of channels X.

To evaluate the system behavior under race conditions, we also define a stress test using five independent infinite-loop processes. Each emits on a unique channel: $A = ch_1! \rightarrow A$, $B = ch_2! \rightarrow B$, $C = ch_3! \rightarrow C$, $D = ch_4! \rightarrow D$, and $E = ch_5! \rightarrow E$. These processes are composed in parallel to form:

$$ComplexRaceCondition = A \parallel B \parallel C \parallel D \parallel E$$

Figure 3.20 shows the corresponding Labeled Transition System (LTS) for both IntegrationDeadlock and ComplexRaceCondition scenarios.

This integration model uses five input channels to simulate scenarios that involve synchronization mismatch, race conditions, and deadlock potential. Channel1 is reused across a simple sequence, a synchronized block, and another sequence to increase contention. Channels 2 and 3 form a nondeterministic choice branch, simulating environmental variability. Channels 4 and 5 are used in parallel executions to test concurrency effects.

This model is designed to assess TorXakis' capability to detect faults that emerge from combined interaction patterns. In particular, it introduces partial synchronization semantics by mismatching the expected number of synchronizations on Channel1. The composition includes a mixture of sequence, choice, and synchronized behaviors, resembling integration of misaligned subsystems. The test assumes no external intervention, meaning any deadlock or failure must arise internally. Furthermore, the model stresses the system using asymmetric channel usage and deliberate omission of full synchronization in certain branches.

For the deadlock scenario, the model tests whether the composed system correctly synchronizes on shared channels. If one or more synchronization expectations are not met, the result is a detectable deadlock, validating TorXakis' ability to handle improper coordination.

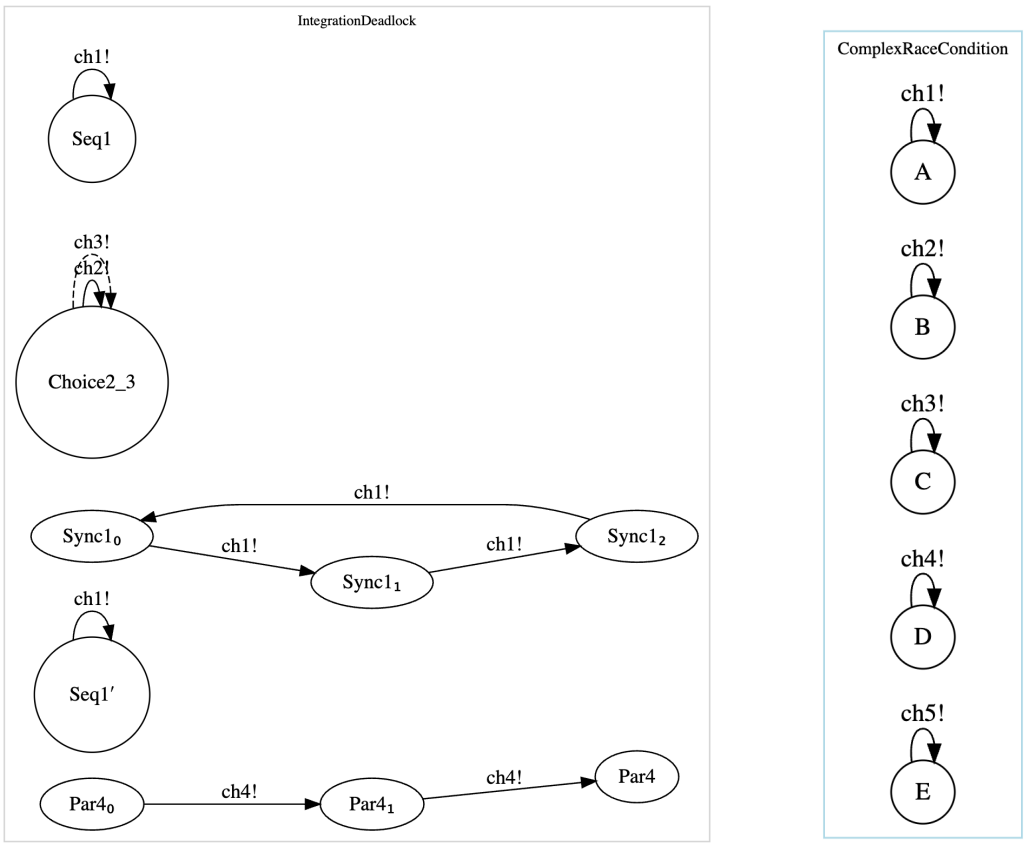

Figure 3.20: LTS model of the *IntegrationDeadlock* and *ComplexRaceCondition* processes.

In the race condition test, the model ensures that all five processes run concurrently without interference. If one process dominates or prevents others from proceeding, the tool should detect it as a fault. Otherwise, the system must allow fair execution of all parallel actions across independent channels.

The complete TorXakis code for this test is provided in Appendix A.7.

3.6.1 Test Outcomes

The Integration Test produced the following key findings:

The system correctly identified and reported a deadlock when one of the synchronizing processes was disabled. This validates the model's ability to detect unresolved synchronization issues. The output of deadlocks in the integration test is shown in the figure 3.21.

```
TXS >>  TorXakis :: Model-Based Testing
TXS >>  txsserver starting: "localhost" : 65054
TXS >>  Solver "z3" initialized : Z3 [4.8.7]
TXS >>  TxsCore initialized
TXS >>  LPEOps version 2019.07.05.02
TXS >>  input files parsed:
TXS >>  ["benchmarknew2.txs"]
TXS >>  stepper IntegrationTestDeadlockScenario
TXS >>  Stepper started
TXS >>  step 10
TXS >>  no state or deadlock
TXS >>  FAIL: No Output (Quiescence)
```

Figure 3.21: The Output of Deadlocks in the Integration Test

During the execution of concurrent components in the integration test, nondeterministic behavior was observed. The order of actions on Channel2 and Channel3 varied across runs. This variation is expected and reflects the inherent nondeterminism of concurrent models, not necessarily a race condition. The output of this behavior in the integration test is shown in Figure 3.22. This output demonstrates a sequence of actions on different channels without a predictable order.

In nondeterministic scenarios, the order of execution can vary between runs because no strict synchronization is imposed between processes. Here, we can see actions on 'Channel1', 'Channel2', 'Channel3', and 'Channel4' in a non-sequential, interleaved order (e.g., 'Channel4', 'Channel2', 'Channel3', etc.). Each channel action '`Act { { ( ChannelX, [] ) } }`' occurs as processes execute independently, and this order may differ in each run of the test due to the concurrent nature of the system. We also repeated this test with the same setup ('step 10'), and we saw a different sequence of channel activations each time, even though no specific order is imposed.

The model successfully detected type mismatches on Channel4 and Channel5, reporting errors when incorrect data types were sent. This demonstrates the robustness of the system in enforcing data integrity. The output of incorrect message passing in the integration test is shown in Figure 3.23.

Our BranchingErrorTest code is well-suited for testing branching errors because it is designed to verify how the process ('choiceProcess') handles different inputs from multiple channels and ensures it transitions to the correct branches ('successBranch' or 'errorBranch') based on the input values. The output of branching errors in the integration test is shown in Figure 3.24. Valid inputs (5 on Channel2, 10 on

```
TXS >>  TorXakis :: Model-Based Testing
TXS >>  txsserver starting: "localhost" : 58396
TXS >>  Solver "z3" initialized : Z3 [4.8.7]
TXS >>  TxsCore initialized
TXS >>  LPEOps version 2019.07.05.02
TXS >>  input files parsed:
TXS >>  ["benchmarknew2.txs"]
TXS >>  stepper ComplexRaceConditionTest
TXS >>  Stepper started
TXS >>  step 10
TXS >>  .....1: Act { { ( Channel11, [] ) } }
TXS >>  .....2: Act { { ( Channel4, [] ) } }
TXS >>  .....3: Act { { ( Channel3, [] ) } }
TXS >>  .....4: Act { { ( Channel5, [] ) } }
TXS >>  .....5: Act { { ( Channel4, [] ) } }
TXS >>  .....6: Act { { ( Channel3, [] ) } }
TXS >>  .....7: Act { { ( Channel5, [] ) } }
TXS >>  .....8: Act { { ( Channel12, [] ) } }
TXS >>  .....9: Act { { ( Channel3, [] ) } }
TXS >>  ....10: Act { { ( Channel3, [] ) } }
TXS >>
TXS >>  PASS
TXS >>
TXS >>  step 10
TXS >>  .....1: Act { { ( Channel5, [] ) } }
TXS >>  .....2: Act { { ( Channel13, [] ) } }
TXS >>  .....3: Act { { ( Channel5, [] ) } }
TXS >>  .....4: Act { { ( Channel4, [] ) } }
TXS >>  .....5: Act { { ( Channel4, [] ) } }
TXS >>  .....6: Act { { ( Channel5, [] ) } }
TXS >>  .....7: Act { { ( Channel12, [] ) } }
TXS >>  .....8: Act { { ( Channel5, [] ) } }
TXS >>  .....9: Act { { ( Channel4, [] ) } }
TXS >>  ....10: Act { { ( Channel2, [] ) } }
TXS >>
TXS >>  PASS
```

Figure 3.22: The Output of Nondeterministic Behavior in the Integration Test

```
TXS >>  TorXakis :: Model-Based Testing
TXS >>  txsserver starting: "localhost" : 59503
TXS >>  Undefined Process at <no location>: [TypeMismatch at line 1422
and column 5: Mismatch in defined (0) and actual (1) channel parameters.
, TypeMismatch at line 1427 and column 5: Mismatch in defined (0) and
actual (1) channel parameters., TypeMismatch at line 1432 and column 5:
Mismatch in defined (0) and actual (1) channel parameters.,TypeMismatch
at line 1437 and column 5: Mismatch in defined (0) and actual (1)
channel parameters.]
TXS >>
TXS >>  CallStack (from HasCallStack):
TXS >>  error, called at src/TorXakis/Compiler.hs:175:25 in
TXS >>  txs-compiler-0.1.0.0-NeT1WHE4z77rZeP6s63kU:TorXakis.Compiler
TXS >>
```

Figure 3.23: The Output of Incorrect Message Passing in the Integration Test

Channel3) transitioned to successBranch as expected, while invalid inputs transitioned to errorBranch. The results confirm that the model handles branching logic correctly and rejects invalid inputs.

The ‘choiceProcess‘ listens for inputs on two channels: ‘Channel2‘ and ‘Channel3‘. Based on the values received, it transitions to either ‘successBranch‘ or ‘errorBranch‘. If ‘Channel2‘ receives ‘5‘, the process transitions to ‘successBranch‘. If ‘Channel3‘ receives ‘10‘, the process transitions to ‘successBranch‘. On the other hand, for any other value on ‘Channel2‘ or ‘Channel3‘ (invalid input), the process transitions to ‘errorBranch‘.

A branching error occurs when the process does not transition correctly based on the input. Our model tests for such errors by sending valid values (‘5‘ or ‘10‘) to test whether ‘choiceProcess‘ correctly transitions to ‘successBranch‘, and sending invalid values (such as ‘99‘) to test whether ‘choiceProcess‘ transitions to ‘errorBranch‘.

The model runs ‘choiceProcess‘ in parallel with two processes namely, sendExpectedValueChannel2 (which sends valid values (‘5‘) on Channel2) and sendUnexpectedValueChannel3 (which sends invalid values (‘99‘) on Channel3). This parallelism creates a realistic test environment where valid and invalid inputs are interleaved, allowing us to observe how the process handles concurrent inputs. Note that all processes in the BEHAVIOUR section start concurrently and so there is no guaranteed order for which process acts first.

```
TXS >>  TorXakis :: Model-Based Testing
TXS >>  txsserver starting: "localhost" : 53582
TXS >>  Solver "z3" initialized : Z3 [4.8.7]
TXS >>  TxsCore initialized
TXS >>  LPEOps version 2019.07.05.02
TXS >>  input files parsed:
TXS >>  ["BranchingError.txs"]
TXS >>  stepper BranchingErrorTest
TXS >>  Stepper started
TXS >>  step 20
TXS >>  .....1: Act { { ( Channel3, [ 10 ] ) } }
TXS >>  .....2: Act { { ( Channel13, [ 99 ] ) } }
TXS >>  .....3: Act { { ( Channel2, [ 5 ] ) } }
TXS >>  .....4: Act { { ( Channel3, [ 99 ] ) } }
TXS >>  .....5: Act { { ( Channel3, [ 99 ] ) } }
TXS >>  .....6: Act { { ( Channel3, [ 99 ] ) } }
TXS >>  .....7: Act { { ( Channel2, [ 5 ] ) } }
TXS >>  .....8: Act { { ( Channel3, [ 99 ] ) } }
TXS >>  .....9: Act { { ( Channel3, [ 99 ] ) } }
TXS >>  ....10: Act { { ( Channel1, [ 1 ] ) } }
TXS >>  ....11: Act { { ( Channel2, [ 5 ] ) } }
TXS >>  ....12: Act { { ( Channel2, [ -90 ] ) } }
TXS >>  ....13: Act { { ( Channel2, [ 5 ] ) } }
TXS >>  ....14: Act { { ( Channel1, [ 99 ] ) } }
TXS >>  ....15: Act { { ( Channel3, [ 99 ] ) } }
TXS >>  ....16: Act { { ( Channel2, [ 5 ] ) } }
TXS >>  ....17: Act { { ( Channel1, [ 99 ] ) } }
TXS >>  ....18: Act { { ( Channel1, [ 99 ] ) } }
TXS >>  ....19: Act { { ( Channel1, [ 99 ] ) } }
TXS >>  ....20: Act { { ( Channel1, [ 99 ] ) } }
TXS >>
TXS >>  PASS
```

Figure 3.24: The Output of Branching Errors in the Integration Test

Chapter 4

On Further Expanding the TorXakis Test Suite

In the previous chapters, various performance evaluation tests were conducted on TorXakis, focusing on aspects such as concurrency, fault tolerance, scalability, resource utilization, model verification, and integration. However, certain tests could not be fully implemented due to intrinsic limitations in TorXakis. This chapter presents a conceptual expansion of these four tests: Real-time Performance Test, User Experience Test, Model Maintainability Test, Security Test.

Unlike the tests in Chapter 4, which were executed and analyzed within the TorXakis environment, the tests in this chapter are theoretical and explore the limitations, challenges, and potential solutions for conducting such evaluations. The goal of this expansion is to propose alternative approaches and lay the groundwork for future research where these tests can be implemented using alternative model-based testing tools or hybrid frameworks. By structuring the discussion in this way, this chapter provides a systematic analysis of the missing test scenarios, offering insights into how they could be realized in future work.

4.1 Real-time Performance Test

Real-time performance testing evaluates how a system behaves under time-constrained scenarios, ensuring that tasks are executed within specific timeframes. However, TorXakis does not support real-time testing or hybrid systems, as confirmed in its official documentation. The documentation explicitly states: "TorXakis supports state & data but no probabilities, real-time, or hybrid systems." ([38], under "Model-based testing tools"). This limitation makes it impossible to perform real-time performance tests directly within the TorXakis framework.

4.1.1 Challenges in Real-Time Testing with TorXakis

The removal of the TIMEOUT function in TorXakis highlights a lack of native support for handling time-sensitive scenarios. This poses several challenges:

1. Inability to Simulate Time-Dependent Behaviors. Real-time systems often require processes to execute within strict time constraints. Without built-in time-handling capabilities, it becomes infeasible to simulate these behaviors in TorXakis.

2. No Support for Real-Time or Hybrid Systems. Hybrid systems, which combine discrete and continuous behaviors (e.g., processes dependent on both state transitions and time constraints), cannot be modeled or tested in TorXakis.

3. Limitations for Performance Verification Performance testing often involves measuring the system's response times or ensuring that tasks complete before a specified deadline. The lack of time-based constructs prevents TorXakis from verifying these properties.

Given TorXakis's limitations, exploring alternative tools specialized for real-time system verification is essential when working on real-time applications. One such tool is UPPAAL, a widely used model checker designed for real-time and hybrid systems. UPPAAL provides robust support for time constraints, verification of real-time properties, and simulating real-time behaviors. However, employing UPPAAL or similar tools falls outside the scope of this thesis, as it focuses on the capabilities and self-testing of TorXakis. Exploring these tools would require a separate evaluation and integration effort, making it more suitable for projects dedicated to real-time system analysis.

4.1.2 Implications of TorXakis's Limitations

While the absence of real-time testing capabilities in TorXakis may seem like a drawback, it reflects the tool's focus on model-based testing for state and data-driven systems. TorXakis is well-suited for applications where timing is not a critical factor, such as testing state transitions or validating data interactions. That is, TorXakis targets non-real-time applications. By excluding real-time features, TorXakis simplifies its scope, making it easier to apply in scenarios where time constraints are not required. Its focus is on simplicity and general use cases. While TorXakis alone cannot handle real-time systems, combining it with complementary tools like UPPAAL could enable a broader range of testing capabilities.

In conclusion, the lack of real-time support in TorXakis is a known limitation that restricts its application in time-sensitive scenarios. The removal of the TIMEOUT function and the absence of hybrid system support further emphasize this constraint. While tools like UPPAAL can address these gaps, they fall outside the

scope of this thesis, which is focused on TorXakis. Nonetheless, this limitation highlights the importance of selecting the appropriate tools for specific testing requirements and paves the way for future explorations into hybrid or integrated testing solutions.

4.2 User Experience Test

The User Experience Test (UX Test) evaluates how intuitive, accessible, and efficient the TorXakis system is for its end-users. The primary focus is to assess how users interact with the TorXakis environment, including its commands, error messages, and overall usability. This test aims to identify barriers that hinder productivity and recommend improvements for a more seamless user experience.

Ease of use examines how straightforward it is for users to navigate and utilize the TorXakis interface.Key questions include: Is the command structure logical and user-friendly? Can users easily perform essential tasks such as creating, testing, and debugging models without requiring extensive training or documentation?

The test assesses whether the workflow facilitates smooth transitions between various stages of model-based testing. Effective error messages are vital for user productivity.This test evaluates whether the system provides informative and actionable feedback when errors occur. Criteria include specificity (are the error messages clear enough to guide users toward resolving issues) and usability (are the messages too technical or generic to be useful for non-expert users).

The learning curve refers to the time and effort required for new users to become proficient with TorXakis. A steep learning curve can deter users from fully utilizing the system's capabilities. This test identifies how accessible TorXakis is for beginners and whether there are sufficient resources (e.g., documentation, tutorials) to support them. Customization allows users to tailor the environment to their specific needs, improving efficiency and satisfaction. The test evaluates: The flexibility of debugging and testing options. Whether users can adapt model definitions for diverse use cases. Consistency ensures that commands and features behave predictably, reducing frustration. This evaluation identifies inconsistencies in the system's behavior that might confuse users.

4.2.1 Challenges Identified

Many users report that TorXakis error messages are overly technical, making it challenging for non-expert users to diagnose and resolve issues. This complexity can lead to delays and reduce productivity during the model development process.

For users new to model-based testing or formal methods, the TorXakis interface and workflows can be intimidating. While the system offers powerful functionality, it often requires significant effort to master, particularly due to limited interactive support or tutorials.

4.2.2 Recommendations

Enhance error messages with detailed explanations of issues. Suggestions for resolving common errors. Contextual guidance for beginners to reduce confusion. Introduce step-by-step tutorials or an interactive mode that walks users through common tasks, such as creating a model, debugging errors, or performing tests. These tutorials could include examples, visual aids, and real-time guidance. Simplify workflows to reduce friction for beginners. Provide shortcuts or templates for common tasks to make the system more accessible. Standardize the behavior of commands and features to eliminate irregularities that might frustrate users. Ensure uniformity in the design and output of error messages, logs, and commands.

To conclude, the User Experience Test highlights areas where TorXakis can improve its accessibility and ease of use, particularly for new users. Key challenges include overly complex error messages and a steep learning curve, which may deter non-expert users. However, with improvements in error messaging, the addition of interactive tutorials, and a focus on consistency, TorXakis can significantly enhance its user experience. These recommendations aim to make the system more approachable, efficient, and satisfying for both beginners and advanced users. By prioritizing usability alongside functionality, TorXakis can strengthen its position as a powerful yet user-friendly tool for model-based testing.

4.3 Model Maintainability Test

The Model Maintainability Test evaluates how easily models in TorXakis can be maintained, modified, and extended over time. Maintainability is a critical aspect of long-term system development, ensuring that models remain adaptable as requirements evolve and systems grow. This test assesses the key factors influencing the manageability of models, aiming to identify challenges and recommend improvements.

Modularity refers to the design of models as smaller, reusable components. This test evaluates whether models are modular enough to allow individual components to be updated or replaced without impacting the overall system. Modular design promotes maintainability by isolating changes to specific parts of the model, reducing the risk of introducing errors.

Readability ensures that models are easy to understand, not only for the original developer but also for others who may work on them in the future. This test examines whether the code is well-structured and whether sufficient comments and explanations are included to convey the purpose and functionality of each part of the model.

As systems grow, models must accommodate new features, processes, or channels without requiring significant restructuring. This test evaluates whether the model's design supports scalability, ensuring that expansions can be made with

minimal technical debt.

Over time, models may need to evolve due to new requirements or corrections. This test assesses how easily models can be modified or extended while preserving existing functionality and minimizing disruptions. Consistency in design patterns ensures predictability and simplifies troubleshooting. This test evaluates whether models follow uniform logical flows and patterns, reducing the cognitive load for developers working on the system.

4.3.1 Challenges Identified

In some cases, models in TorXakis lack uniformity in structure. Developers may use varied styles or approaches, leading to inconsistencies that complicate maintenance. As models become more complex, synchronizing multiple processes across channels can result in increased complexity. This can make it harder to expand or modify models effectively. Many TorXakis models lack detailed comments or explanations. Without sufficient documentation, revisiting older models requires developers to re-learn the system, slowing down the process of making updates or fixes.

4.3.2 Recommendations

Break down models into smaller, self-contained processes that can be reused and tested independently. Modular models are easier to maintain and scale, allowing developers to make updates without affecting unrelated parts of the system. Adopt consistent design patterns across all models to improve predictability and reduce errors. This practice ensures that new features can be added seamlessly, maintaining uniformity throughout the system. Include detailed comments in every model, explaining its structure, purpose, and logic. Well-documented models enhance maintainability by reducing the time required for future developers to understand and modify them.

In conclusion, the Model Maintainability Test emphasizes the importance of creating models that remain adaptable and manageable as the system evolves. The evaluation highlights areas where TorXakis could improve, such as adopting a more modular design, enforcing consistent patterns, and enhancing documentation. By addressing these challenges, TorXakis can ensure that its models remain scalable, predictable, and easy to update, supporting long-term system growth and sustainability. These improvements would not only streamline the development process but also make the system more accessible to new developers, ensuring the continued success of TorXakis in diverse applications.

4.4 Security Test

The Security and Reliability Test for TorXakis assesses the system's ability to maintain data integrity, robustness, and stability under various conditions. While TorXakis is not a public-facing tool, ensuring consistent data exchange, controlled access, and reliable error handling is essential, particularly in collaborative environments where multiple users modify models and test configurations.

Since TorXakis lacks built-in security features, this conceptual evaluation focuses on data consistency, input validation, channel synchronization, and traceability. The study explores how system behavior might be affected by unexpected input values, high-load stress conditions, and logging mechanisms. Theoretical scenarios were developed to analyze how TorXakis would handle data inconsistencies, malformed inputs, and desynchronization under stress.

4.4.1 Findings and Challenges

Conceptual analysis indicates that desynchronization may occur under high-load conditions, affecting channel communication stability. While TorXakis flags malformed inputs, the error messages may be too technical, limiting their usefulness for diagnosing failures. The lack of structured access control mechanisms poses risks in multi-user settings, potentially allowing unintended modifications. Logging mechanisms, while functional, may not capture sufficient detail for effective debugging and traceability, particularly in analyzing failures under stress conditions.

4.4.2 Recommendations

Strengthening synchronization protocols could improve system stability under high loads. Refining error messages to provide clearer diagnostics would enhance usability. Implementing user roles and permissions would prevent unintended modifications in collaborative environments, and improving logging mechanisms could facilitate more effective debugging and system traceability.

Ensuring robust security and reliability in TorXakis is essential for maintaining test accuracy and system integrity. Addressing potential synchronization issues, improving input validation, and refining logging mechanisms would enhance usability and maintainability. While these security aspects may not be critical for single-user implementations, they become increasingly important in large-scale, collaborative testing environments.

Chapter 5

Conclusion and Future Work

This thesis set out to evaluate the central research question of whether TorXakis is a correct implementation of the ioco testing theory. To answer this, a comprehensive set of empirical tests was designed and conducted, each targeting key behavioral aspects of model-based systems, including concurrency, fault tolerance, scalability, resource utilization, semantic correctness, and integration performance. Through these tests, we examined whether TorXakis's observable behavior and output responses aligned with the semantic expectations defined by the ioco theory.

The concurrency and deadlock tests demonstrated TorXakis's ability to simulate high-load parallel systems while maintaining consistency in trace outputs and avoiding system-level deadlocks. These tests confirmed that TorXakis can successfully manage up to 19 concurrent processes, preserving synchronization and progress, which are essential elements of ioco conformance. Deadlock scenarios were accurately detected using intentional circular dependencies between channels, further validating the tool's ability to reflect liveness and progress properties. While the detection of non-deterministic issues such as race conditions was not conclusive, the test still revealed valuable insights about the tool's handling of realistic concurrency challenges.

The fault tolerance test shifted focus from validating systems to evaluating TorXakis itself as a fault-resilient testing tool. When subjected to simulated failures such as channel disruptions, nondeterministic decisions, and recovery sequences, the tool exhibited stable behavior and consistent outputs. TorXakis managed to handle injected faults without crashing, producing meaningful results and maintaining operational stability. These outcomes suggest that the tool is not only capable of simulating fault-tolerant systems but is also structurally robust against faults within its own test environments.

The scalability and resource utilization tests provided further insight into the tool's limitations and strengths under growing model complexity. While TorXakis performed well up to a defined threshold of 19 channels, pushing beyond this

point revealed exponential increases in CPU and memory usage. These performance constraints stemmed from solver inefficiencies and the lack of optimization in managing nested parallelism and synchronized processes. Nonetheless, within its operational boundaries, the system maintained trace integrity and did not produce faulty outputs, supporting partial conformance to ioco expectations in scalable environments.

Model verification tests played a crucial role in evaluating TorXakis's semantic accuracy. Various correctness properties such as liveness, deadlock-freedom, synchronization, data integrity, choice handling, and concurrent behavior were systematically validated. The tool effectively captured faults like infinite waiting, deadlocks, incorrect synchronization, invalid data propagation, and decision-making inconsistencies. These tests confirmed that TorXakis is capable of accurately modeling and verifying behaviors in accordance with the theoretical properties defined by ioco, reinforcing its credibility as a conformance testing tool. This test is a good basis for the development of a larger test suite that would establish the correctness of TorXakis.

Integration tests further extended the assessment by simulating complex real-world scenarios not typically covered by standard functional tests. These included edge cases such as partial synchronization failures, race conditions, incorrect message passing, and branching logic errors. TorXakis successfully detected and reported such faults, demonstrating its capability to handle system-level interactions and deviations that are critical for robust model validation. These integration scenarios added practical depth to the test suite, enhancing its coverage and applicability for production-grade systems.

In conclusion, this thesis has demonstrated that TorXakis exhibits a strong and meaningful alignment with the ioco testing theory across a variety of model-based scenarios. While formal verification was beyond the scope of this study, the empirical evidence supports the claim that TorXakis implements the core principles of ioco correctly. The tool's strengths in trace observation, fault handling, and behavioral verification show that it can be relied upon for systematic testing tasks that reflect theoretical conformance expectations. TorXakis accurately simulates expected observable behaviors, handles refusals and quiescence conditions, and responds predictably to faults and concurrency challenges. Although certain architectural limitations—such as limited scalability, lack of native real-time capabilities, and usability barriers—affect its applicability in highly complex or time-sensitive environments, these constraints do not undermine its foundational conformance. Rather, they highlight the need for future research to extend the tool's temporal expressiveness, optimize its performance for larger and more dynamic models, and pursue formal verification of its internal mechanisms against ioco semantics. Overall, TorXakis emerges from this evaluation as a practically ioco-compliant tool, capable of supporting model-based testing with meaningful accuracy and robustness, and positioned as a solid foundation for continued development and formalization.

5.1 Some Answers to Our Research Question

Our results provide a few answers to the question of TorXakis correctness. By executing a diverse suite of custom-designed test models——including those simulating concurrency, deadlocks, nondeterministic behavior, fault tolerance, and integration——we observed how TorXakis responds under controlled and theoretically meaningful conditions. The outputs consistently showed behavioral patterns that match the ioco semantics in many core areas, particularly in how traces are handled, how synchronization is performed, and how faults are tolerated within structured models. We thus argue that for practical purposes TorXakis is a good implementation of ioco.

The analysis also highlighted limitations in certain domains, such as real-time responsiveness and scalability under high process/channel loads. These observations expose areas that warrant further improvement or formal verification. By documenting these strengths and gaps, this research fulfills its intended goal of providing a structured framework for assessing conformance in model-based testing tools. The outcomes thus validate both the relevance of the research question and the effectiveness of the empirical methodology used to address it.

Recall now that our ultimate thesis was that TorXakis can be formally verified using TorXakis itself. The empirical evidence gathered throughout this paper is mixed. On one hand, we walked TorXakis through a reasonably comprehensive test suite and did not observe any anomalies. We can conclude tentatively that the tool works as expected. On the other hand, one can reasonably expect that the test scenarios that will be eventually developed for a full formal verification will be quite complex. Our empirical evidence shows that it may not be possible to run those tests. Indeed, we have noted an exponential increase in resource requirements, which for the time being makes the possibility of running very complex tests doubtful. Further investigations are definitely needed.

5.2 Future Work and Practical Recommendations

To further investigate the central research question (whether TorXakis a correct implementation of ioco), future work should focus on addressing the limitations identified in this study and in particular improving the tool's scalability.

Based on the insights gained from empirical testing, we can propose several improvements to enhance the practical performance of TorXakis and also work toward establishing conformance to ioco semantics. The biggest such an improvement is in scalability. The system's capacity to handle concurrent processes and channels is notably limited, with failure occurring beyond 19 channels due to architectural and resource limitations. Refactoring the system architecture to support dynamic scaling, potentially through distributed processing or parallel execution techniques should be a priority. This would allow TorXakis to handle more complex ioco

models involving multiple interacting components, enhancing its applicability for large-scale testing. At the same time, a deeper investigation into the complexity of the ioco algorithms should take place, with the goal of determining to what degree implementation changes can improve performance.

The absence of constructs like TIMEOUT limits the tool's ability to model time-constrained systems. One should consider the reintroduction of real-time constructs, eventually implementing timed ioco. This would enable the evaluation of systems like embedded controllers and real-time protocols that rely on timing constraints as part of their conformance behavior.

On a very practical note we found that the tool's error messages are overly technical and often lack actionable guidance, making it difficult for users to debug or interpret unexpected behavior. Enhancing diagnostic feedback by providing user-friendly error descriptions and contextual recommendations is worth considering. Clearer insights would support better trace analysis, helping users validate whether the system behavior conforms to ioco semantics. Additionally, the current logging mechanism does not provide detailed traces for in-depth behavioral analysis. A robust audit logging system to capture observable actions, errors, and process synchronization outcomes would aid in evaluating whether execution traces align with the expected ioco-compliant behavior.

Finally, we noted that a steep learning curve and insufficient documentation create barriers for new users and researchers. We would like to see comprehensive tutorials and interactive examples covering model creation, testing strategies, and ioco-specific interpretations. Better onboarding resources will broaden the user base and make the tool more accessible to non-experts.

5.2.1 Recommendations for Researchers and Practitioners

Future researchers and practitioners can build upon this work to both improve TorXakis and develop more reliable model-based testing frameworks aligned with formal semantics. The following directions are suggested:

- Investigate distributed and cloud-based infrastructures to overcome hardware limitations and enable large-scale testing of high-concurrency ioco models.
- Explore the integration of TorXakis with formal tools such as UPPAAL to support hybrid specifications.
- Extend the modeling scope to real-time behavior, thus implementing timed ioco.
- Develop optimization strategies for handling parallel test execution, dynamic prioritization of channels, and runtime resource allocation to improve performance and responsiveness.

Last but not least, the possible specifications that establish TorXakis as a formally correct implementation of ioco should definitely be pursued. As noted above however, this might not be possible. In the event that TorXakis proves unable to run such a testing scenario on itself (as our research seems to suggest), a reasonable substitute may be the use of AI-based techniques to provide a reasonable guarantee of correctness instead. Recent advancements in MBT have led to the integration of artificial intelligence and machine learning techniques to enhance test case generation and selection. AI-driven MBT approaches aim to optimize testing by predicting critical system behaviors, reducing redundant test cases, and improving fault detection rates. Future research in this area is expected to explore the combination of MBT with AI-based adaptive testing frameworks, enabling more efficient and intelligent test execution strategies [28]. Applying such techniques on TorXakis itself may prove to be a practical substitute of a proof of correctness should such a proof turn out not to be possible.

Bibliography

[1] M. Abdolahi, *Grading with TorXakis*, M.Sc. thesis, Dept. of Computer Science, Bishop's University, Sherbrooke, QC, Canada, Aug. 2023.

[2] D. Ahman and M. Kääramees *Constraint-Based Heuristic On-line Test Generation from Non-deterministic I/O EFSMs*, arXiv:1202.6126 , 2012.

[3] B.K. Aichernig, W. Mostowski, M.R. Mousavi, M. Tappler, and M. Taromirad, *Model Learning and Model-Based Testing*, in A. Bennaceur, R. Hähnle, K. Meinke (eds) Machine Learning for Dynamic Software Analysis: Potentials and Limits, Springer LNCS 11026.

[4] P. Ammann and J. Offutt, *Introduction to Software Testing*, 1st ed., Cambridge University Press, 2008.

[5] J.C.M. Baeten, T. Basten, and M.A. Reniers, *Process Algebra: Equational Theories of Communicating Processes*. Cambridge University Press, 2010.

[6] O. Balci, *Verification, Validation, and Testing Techniques*, *ACM SIGSIM*, 2016.

[7] M. van der Bijl, and F. Peureux, *I/O-automata Based Testing*, in Model-Based Testing of Reactive Systems: Advanced Lectures, M. Broy, B. Jonsson, J.P. Katoen, M. Leucker, and A. Pretschner, Eds., Springer LNCS 3472, 2005, pp. 173–200.

[8] BrowserStack, *What is Model-Based Testing in Software Testing*, BrowserStack Guide, Oct 2023.

[9] S.D. Bruda, *Preorder Relations*, in Model-Based Testing of Reactive Systems: Advanced Lectures, M. Broy, B. Jonsson, J.P. Katoen, M. Leucker, and A. Pretschner, Eds., Springer LNCS 3472, 2005, pp. 117–150.

[10] I. Forgács and A. Kovács, *Modern Software Testing Techniques: A Practical Guide for Developers and Testers*. Springer, 2024.

[11] M.-C. Gaudel *Formal Methods for Software Testing*, HAL Archives, 2017.

[12] V. Garous, A.B. Keleş, Y. Balaman, Z.Ö. Güler, and A. Arcuri, *Model-based testing in practice: An experience report from the web applications domain*, Journal of Systems and Software, vol. 180, 2021.

[13] GeeksforGeeks, *Behavior Driven Testing*, May 2022.

[14] GeeksforGeeks, *Fuzz Testing*, 2023.

[15] M. Gleirscher, J. van de Pol, and J. Woodcock, *A Manifesto for Applicable Formal Methods*, Software and Systems Modeling, vol. 22, 2023, pp. 1737–1749.

[16] J.P. Katoen, *Labelled Transition Systems*, in Model-Based Testing of Reactive Systems: Advanced Lectures, M. Broy, B. Jonsson, J.P. Katoen, M. Leucker, and A. Pretschner, Eds., Springer LNCS 3472, pp. 615–616.

[17] LambdaTest, *What is Model-Based Testing: An Overview*, LambdaTest Learning Hub, Feb. 2023.

[18] Mark Utting and Bruno Legeard, *Practical Model-Based Testing: A Tools Approach*, 1st ed., Morgan Kaufmann, 2006.

[19] R. Milner, *A Calculus of Communicating Systems*, Springer LNCS vol. 92, 1980.

[20] S. Misra, *Software Testing Techniques*," in Proceedings of the Canadian Conference on Electrical and Computer Engineering, 2003, pp. 1873–1877.

[21] T. Ostrand, *White-Box Testing*," in Encyclopedia of Software Engineering, vol. 2, J. Marciniak, Ed., Wiley, 2002, pp. 1256–1262.

[22] G. Petrovic, G. Fraser, M. Ivanković, and R. Just, *Practical Mutation Testing at Scale: A View from Google*, IEEE Transactions on Software Engineering, vol. 47, 2021, pp. 2802–2817.

[23] A. Pretschner, W. Prenninger, S. Wagner, C. Kühnel, M. Baumgartner, B. Sostawa, R. Zölch, and T. Stauner (2005), *One evaluation of model-based testing and its automation*, in Proceedings of the 27th International Conference on Software Engineering (ICSE '05), 2005, pp. 392–401.

[24] S. Polzin, *Understanding Model-Based Testing: Benefits, Challenges, and Use Cases*, Qt Blog, 2023.

[25] S. Rädler, L. Berardinelli, K. Winter, A. Rahimi, and S. Rinderle-Ma, *Bridging MDE and AI: A Systematic Review of Domain-Specific Languages and Model-Driven Practices in AI Software Systems Engineering*, Software and Systems Modeling, vol. 23, 2024, pp. 1024–1045.

[26] A. W. Roscoe, *The Theory and Practice of Concurrency*. Prentice Hall, 1997.

[27] A. Sanchez, P. Delgado-Perez, I. Medina-Bulo, and S. Segura, *Mutation Testing in the Wild: Findings from GitHub*, Empirical Software Engineering, vol. 27, no. 3, 2022.

[28] H. Sartaj, A. Muqeet, M.Z. Iqbal, and M.U. Khan, *Automated System-level Testing of Unmanned Aerial Systems*, arXiv:2403.15857, 2024.

[29] C. Meadows, *Program Verification and Security*, in H.C.A van Tilborg, S. Jajodia (eds), Encyclopedia of Cryptography and Security, Springer, 2011.

[30] G. Sypolt, *The Challenges and Benefits of Model-Based Testing*, Sauce Labs, 2017.

[31] A. Talreja, *What is Exploratory Testing? Learn with a Real World Example*, *Master Software Testing*, Apr. 2024.

[32] *TorXakis: A Tool for Model-Based Testing*, retrieved Apr. 2025.

[33] *TorXakis: Model-Based Testing Tool*, GitHub repository.

[34] TorXakis: TestBench, retrieved Apr. 2025.

[35] J. Tretmans, *Conformance Testing with Labelled Transition Systems: Implementation Relations and Test Generation*, Computer Networks and ISDN Systems, vol. 29, no. 1, 1996, pp. 49—79.

[36] J. Tretmans, *Model-Based Testing with Labelled Transition Systems*, in Formal Methods and Testing, R. M. Hierons, J. P. Bowen, and M. Harman, Eds., Springer LNCS 4949, 2008, pp. 1–38.

[37] J. Tretmans and P. van der Laar, *Model-Based Testing with TorXakis: The Mysteries of Dropbox Revisited*, in Proc. 30th CECIIS, Varaždin, Croatia, Oct. 2019.

[38] J. Tretmans, *TorXakis: A Model-Based Testing Tool*, TorXakis Documentation, 2020.

[39] V. Tschaen, *Test Generation Algorithms Based on Preorder Relations*, in Model-Based Testing of Reactive Systems: Advanced Lectures, M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner, Eds., Springer LNCS 3472, 2005, pp. 151–172.

Appendix A

TorXakis Model Code Listing

A.1 Maximum Concurrency Stress Test

```
CHAN IN
CHAN OUT   Channel1, Channel2, Channel3, Channel4, Channel5,
Channel6, Channel7, Channel8, Channel9, ChannelInt1, ChannelInt2,
ChannelInt3, Channel10Ints, Channel10Ints_b
BEHAVIOUR
    (
        -- Parallel sequences with all channels
        sequence [ Channel1 ] ( )
    |||
        sequence [ Channel2 ] ( )
    |||
        sequence [ Channel3 ] ( )
    |||
        sequence [ Channel4 ] ( )
    |||
        sequence [ Channel5 ] ( )
    |||
        sequence [ Channel6 ] ( )
    |||
        sequence [ Channel7 ] ( )
    |||
        sequence [ Channel8 ] ( )
    |||
        sequence [ Channel9 ] ( )
    )
|[ Channel1, Channel2, Channel3, Channel4, Channel5, Channel6,
Channel7, Channel8, Channel9 ]|
```

```
    (
        -- Synchronized sequences with interleaving steps
        synchronizedN [ Channel1 ] ( 5 )
    |||
        synchronizedN [ Channel2 ] ( 5 )
    |||
        synchronizedN [ Channel3 ] ( 5 )
    |||
        synchronizedN [ Channel4 ] ( 5 )
    |||
        synchronizedN [ Channel5 ] ( 5 )
    |||
        synchronizedN [ Channel6 ] ( 5 )
    |||
        synchronizedN [ Channel7 ] ( 5 )
    |||
        synchronizedN [ Channel8 ] ( 5 )
    |||
        synchronizedN [ Channel9 ] ( 5 )
    )
|[ ChannelInt1, ChannelInt2, ChannelInt3 ]|
    (
        -- Parallel data sequences with integer channels
        parallelDataN [ ChannelInt1 ] ( 10 )
    |||
        parallelDataN [ ChannelInt2 ] ( 10 )
    |||
        parallelDataN [ ChannelInt3 ] ( 10 )
    )
|[ Channel10Ints, Channel10Ints_b ]|
    (
        -- Sequences handling complex data structures
        sequence10Ints [ Channel10Ints ] ( )
    |||
        sequence10Ints_b [ Channel10Ints_b ] ( )
    )
|[ Channel1, Channel2, Channel3, Channel4, Channel5, Channel6,
Channel7, Channel8, Channel9, ChannelInt1, ChannelInt2, ChannelInt3,
Channel10Ints, Channel10Ints_b ]|
    (
-- Nested and interleaved parallel and synchronized processes
        synchronizedSequences [ Channel1 ] ( )
```

```
        |||
            synchronizedSequences [ Channel2 ] ( )
        |||
            synchronizedSequences [ Channel3 ] ( )
        |||
            synchronizedSequences [ Channel4 ] ( )
        |||
            synchronizedSequences [ Channel5 ] ( )
        |||
            synchronizedSequences [ Channel6 ] ( )
        |||
            synchronizedSequences [ Channel7 ] ( )
        |||
            synchronizedSequences [ Channel8 ] ( )
        |||
            synchronizedSequences [ Channel9 ] ( )
        )
ENDDEF
```

A.2 Deadlock Injection and Detection Evaluation

The implementation involves defining a process called ‘deadlockSequence‘, where each instance of this process attempts to communicate on a specific channel and then stops (‘STOP‘). For example, process ‘A‘ might wait on ‘Channel1‘ while ‘B‘ waits on ‘Channel2‘, with ‘B‘ requiring ‘A‘ to finish before it can proceed. However, ‘A‘ also waits on another channel, creating a dependency cycle. By organizing these ‘deadlockSequence‘ instances in a synchronized composition set, they collectively enter a state where no process can continue because each is waiting on the next. Multiple ‘deadlockSequence‘ processes are started in parallel and are synchronized on specific channels. This synchronization enforces the dependency between processes and prevents any one process from executing without the others. The cyclic dependency formed by synchronizing ‘deadlockSequence‘ processes on interconnected channels is what generates the deadlock.

Here is the complete model definition:

```
PROCDEF deadlockSequence [ Channel1, Channel2 ] ( ) ::=
    Channel1 >-> STOP  -- Deadlock after Channel1
##
    Channel2 >-> STOP  -- Deadlock after Channel2
ENDDEF

MODELDEF MaxConcurrencyStressTestWithDeadlocks ::=
    CHAN IN
    CHAN OUT   Channel1, Channel2, Channel3, Channel4, Channel5,
    Channel6, Channel7, Channel8, Channel9, ChannelInt1, ChannelInt2,
    ChannelInt3, Channel10Ints, Channel10Ints_b
    BEHAVIOUR
        (
            -- Parallel sequences
            sequence [ Channel1 ] ( )
        |||
            sequence [ Channel2 ] ( )
        |||
            sequence [ Channel3 ] ( )
        )
    |[ Channel4, Channel5, Channel6 ]|
        (
            -- Synchronized sequences with interleaving steps
            synchronizedN [ Channel4 ] ( 5 )
        |||
            synchronizedN [ Channel5 ] ( 5 )
        |||
```

```
            synchronizedN [ Channel6 ] ( 5 )
        )
    |[ ChannelInt1, ChannelInt2, ChannelInt3 ]|
        (
            -- Parallel data sequences with integer channels
            parallelDataN [ ChannelInt1 ] ( 10 )
        |||
            parallelDataN [ ChannelInt2 ] ( 10 )
        |||
            parallelDataN [ ChannelInt3 ] ( 10 )
        )
    |[ Channel10Ints, Channel10Ints_b ]|
        (
            -- Sequences handling complex data structures
            sequence10Ints [ Channel10Ints ] ( )
        |||
            sequence10Ints_b [ Channel10Ints_b ] ( )
        )
    |[ Channel7, Channel8, Channel9 ]|
        (
    -- Nested and interleaved parallel and synchronized processes
            synchronizedSequences [ Channel7 ] ( )
        |||
            synchronizedSequences [ Channel8 ] ( )
        |||
            synchronizedSequences [ Channel9 ] ( )
        )
    |[ Channel1, Channel2, Channel3, Channel4 ]|
        (
            -- Intentional deadlock sequences
            deadlockSequence [ Channel1, Channel2 ] ( )
        |||
            deadlockSequence [ Channel2, Channel3 ] ( )
        |||
            deadlockSequence [ Channel3, Channel4 ] ( )
        |||
            deadlockSequence [ Channel4, Channel1 ] ( )
        )
ENDDEF
```

A.3 Fault Tolerance Test

```
PROCDEF faultySequence [ PrimaryChannel, BackupChannel ] ( ) ::=
PrimaryChannel >-> faultySequence [ PrimaryChannel, BackupChannel ]( )
##
BackupChannel >-> faultySequence [ PrimaryChannel, BackupChannel ]( )
ENDDEF

###############################
MODELDEF SpecFaultySequence ::=
CHAN IN
CHAN OUT PrimaryChannel, BackupChannel
BEHAVIOUR
faultySequence [ PrimaryChannel, BackupChannel ] ()
ENDDEF

PROCDEF faultyChoice [ Channel1, Channel2 ] ( ) ::=
Channel1 >-> faultyChoice [ Channel1, Channel2 ] ( )
##
Channel2 >-> faultyChoice [ Channel1, Channel2 ] ( )
ENDDEF

################################
MODELDEF SpecFaultyChoice ::=
CHAN IN
CHAN OUT
BEHAVIOUR
faultyChoice [ Channel1, Channel2 ] ()
ENDDEF

PROCDEF combinedFaultTolerance [ PrimaryChannel1, BackupChannel1,
PrimaryChannel2, BackupChannel2, Channel1, Channel2 ] ( ) ::=
faultySequence [ PrimaryChannel1, BackupChannel1 ] ()
|||
faultySequence [ PrimaryChannel2, BackupChannel2 ] ()
|||
faultyChoice [ Channel1, Channel2 ] ()
ENDDEF

###################################
MODELDEF SpecCombinedFaultTolerance ::=
CHAN IN
CHAN OUT PrimaryChannel1, BackupChannel1, PrimaryChannel2,
```

```
BackupChannel2, Channel1, Channel2
BEHAVIOUR
combinedFaultTolerance [ PrimaryChannel1, BackupChannel1,
PrimaryChannel2, BackupChannel2, Channel1, Channel2 ] ()
ENDDEF
```

A.4 Scalability Test

The test utilized the following core constructs: First, we define a couple of processes.

- CombinedConcurrent: This process combines multiple sequences and choice operations over nine channels to simulate a highly concurrent but structured communication environment.
- CombinedScalabilityTest: This process scales up the number of channels to 50, integrating various concurrency patterns such as parallel, synchronized, and fault-tolerant operations.

We then implemented the following concurrency patterns:

- Sequential Operations: Basic sequences were defined for processes to handle ordered communication on individual channels.
- Choice Operations: Processes were designed to make non-deterministic decisions based on input from multiple channels.
- Parallel Operations: Processes such as parallelN and parallelAlternateN were used to simulate concurrent execution across multiple instances.
- Synchronized Operations: Processes like synchronizedN and synchronizedAlternateN ensured proper coordination and synchronization between different channels.
- Fault-Tolerant Processes: Additional processes tested fault tolerance under high concurrency by combining fault-handling mechanisms with communication.

Finally, we define a model to specify the system's input and output channels and bind them to the combined ScalabilityTest process. This ensured a cohesive and scalable structure for testing.

Here is the complete model definition:

```
PROCDEF combinedConcurrent [
    Channel1, Channel2, Channel3, Channel4, Channel5, Channel6,
    Channel7, Channel8, Channel9
] ( ) ::=
    sequence [ Channel1 ] ()
    |||
    sequence [ Channel2 ] ()
    |||
    sequence [ Channel3 ] ()
    |||
```

```
    choice [ Channel4, Channel5 ] ()
    |||
    choice [ Channel6, Channel7 ] ()
    |||
    choice [ Channel8, Channel9 ] ()
ENDDEF


-- Define the process for combined scalability test
PROCDEF combinedScalabilityTest [
Channel1, Channel2, Channel3, Channel4, Channel5, Channel6, Channel7,
Channel8, Channel9, Channel10,Channel11, Channel12, Channel13,
Channel14,Channel15,Channel16, Channel17, Channel18, Channel19,
Channel20,Channel21, Channel22, Channel23, Channel24, Channel25,
Channel26, Channel27, Channel28, Channel29, Channel30,Channel31,
Channel32, Channel33, Channel34, Channel35, Channel36, Channel37,
Channel38, Channel39, Channel40, Channel41, Channel42, Channel43,
Channel44, Channel45, Channel46, Channel47, Channel48, Channel49,
Channel50
] ( ) ::=
(
    sequence [ Channel1 ] ()
|||
    sequence [ Channel2 ] ()
|||
    sequence [ Channel3 ] ()
|||
    choice [ Channel4, Channel5 ] ()
|||
    choice [ Channel6, Channel7 ] ()
|||
    choice [ Channel8, Channel9 ] ()
|||
    parallelN [ Channel10 ] (5)
|||
    parallelN [ Channel11 ] (5)
|||
    parallelAlternateN [ Channel12, Channel13 ] (5)
|||
    parallelAlternateN [ Channel14, Channel15 ] (5)
|||
    synchronizedN [ Channel16 ] (5)
```

```
|||
    synchronizedN [ Channel17 ] (5)
|||
    synchronizedAlternateN [ Channel18, Channel19 ] (5)
|||
    synchronizedAlternateN [ Channel20, Channel21 ] (5)
|||
    synchronizedIStepN [ Channel22 ] (5)
|||
    synchronizedIStepN [ Channel23 ] (5)
|||
    sequenceEnable [ Channel24 ] ()
|||
    sequenceEnable [ Channel25 ] ()
|||
    hideC_synchC_Par_Alternate_C_X [ Channel26 ] (5)
|||
    hideC_synchC_Par_Alternate_C_Xi [ Channel27, Channel28, Channel29,
    Channel30 ] ()
|||
    hideC_synchX_Par_Alternate_C_X [ Channel31 ] (5)
|||
    synchronizedSequences [ Channel32 ] ()
|||
    synchronizedSequences [ Channel33 ] ()
|||
    synchronizedSequences [ Channel34 ] ()
|||
    synchronizedSequences [ Channel35 ] ()
|||
    combinedConcurrent [ Channel36, Channel37,
    Channel38, Channel39, Channel40, Channel41,
    Channel42, Channel43, Channel44 ] ()
|||
    combinedFaultTolerance [ Channel45, Channel46,
    Channel47, Channel48, Channel49, Channel50 ] ()
)
ENDDEF


MODELDEF SpecCombinedScalabilityTest ::=
CHAN IN
```

```
CHAN OUT
Channel1, Channel2, Channel3, Channel4, Channel5, Channel6, Channel7,
Channel8, Channel9, Channel10, Channel11, Channel12, Channel13,
Channel14,Channel15, Channel16, Channel17, Channel18, Channel19,
Channel20,Channel21, Channel22, Channel23, Channel24, Channel25,
Channel26, Channel27, Channel28, Channel29, Channel30,Channel31,
Channel32, Channel33, Channel34, Channel35,Channel36, Channel37,
Channel38, Channel39, Channel40,Channel41, Channel42, Channel43,
Channel44, Channel45,Channel46, Channel47, Channel48, Channel49,
Channel50
BEHAVIOUR
    combinedScalabilityTest [
    Channel1, Channel2, Channel3, Channel4, Channel5, Channel6,
    Channel7, Channel8, Channel9, Channel10,Channel11, Channel12,
    Channel13, Channel14, Channel15, Channel16, Channel17, Channel18,
    Channel19, Channel20, Channel21, Channel22, Channel23, Channel24,
    Channel25, Channel26, Channel27, Channel28, Channel29, Channel30,
    Channel31, Channel32, Channel33, Channel34,Channel35, Channel36,
    Channel37, Channel38,Channel39, Channel40,Channel41, Channel42,
    Channel43, Channel44, Channel45, Channel46, Channel47, Channel48,
    Channel49, Channel50
    ] ()
ENDDEF
```

A.5 Resource Utilization Test

```
MODELDEF ResourceUtilizationTest ::=
    CHAN IN
    CHAN OUT   Channel1, Channel2, Channel3, Channel4, Channel5,
    Channel6, Channel7, Channel8, Channel9,ChannelInt1,
    ChannelInt2, ChannelInt3, Channel10Ints,Channel10Ints_b
    BEHAVIOUR
        (
-- Testing simple sequence performance across multiple channels
            sequence [ Channel1 ] ( )
        |||
            sequence [ Channel2 ] ( )
        |||
            sequence [ Channel3 ] ( )
        |||
            sequence [ Channel4 ] ( )
        |||
            sequence [ Channel5 ] ( )
        |||
            sequence [ Channel6 ] ( )
        |||
            sequence [ Channel7 ] ( )
        |||
            sequence [ Channel8 ] ( )
        |||
            sequence [ Channel9 ] ( )
        )
    |[ Channel1, Channel2, Channel3, Channel4, Channel5, Channel6,
    Channel7, Channel8, Channel9 ]|
        (
            -- Synchronized processes for resource-intensive tasks
            synchronizedN [ Channel1 ] ( 5 )
        |||
            synchronizedN [ Channel2 ] ( 5 )
        |||
            synchronizedN [ Channel3 ] ( 5 )
        |||
            synchronizedN [ Channel4 ] ( 5 )
        |||
            synchronizedN [ Channel5 ] ( 5 )
        |||
```

```
            synchronizedN [ Channel6 ] ( 5 )
        |||
            synchronizedN [ Channel7 ] ( 5 )
        |||
            synchronizedN [ Channel8 ] ( 5 )
        |||
            synchronizedN [ Channel9 ] ( 5 )
        )
    |[ ChannelInt1, ChannelInt2, ChannelInt3 ]|
        (
            -- Testing parallel data handling
            parallelDataN [ ChannelInt1 ] ( 10 )
        |||
            parallelDataN [ ChannelInt2 ] ( 10 )
        |||
            parallelDataN [ ChannelInt3 ] ( 10 )
        )
    |[ Channel10Ints, Channel10Ints_b ]|
        (
            -- Complex data handling tests
            sequence10Ints [ Channel10Ints ] ( )
        |||
            sequence10Ints_b [ Channel10Ints_b ] ( )
        )
    |[ Channel1, Channel2, Channel3, Channel4, Channel5, Channel6,
    Channel7, Channel8, Channel9, ChannelInt1, ChannelInt2,
    ChannelInt3, Channel10Ints, Channel10Ints_b ]|
        (
-- Combined nested, parallel, and synchronized tasks
for maximum resource utilization
            synchronizedSequences [ Channel1 ] ( )
        |||
            synchronizedSequences [ Channel2 ] ( )
        |||
            synchronizedSequences [ Channel3 ] ( )
        |||
            synchronizedSequences [ Channel4 ] ( )
        |||
            synchronizedSequences [ Channel5 ] ( )
        |||
            synchronizedSequences [ Channel6 ] ( )
        |||
```

```
            synchronizedSequences [ Channel7 ] ( )
        |||
            synchronizedSequences [ Channel8 ] ( )
        |||
            synchronizedSequences [ Channel9 ] ( )
        )
ENDDEF
```

A.6 Model Verification Test

```
-- ------------------------------------------------------------
-- 1. Liveness Verification: Ensure that the system progresses
without stalling.
-- ------------------------------------------------------------
-- This test verifies that the sequence continues without getting
stuck.
PROCDEF verifyLiveness [ Channel ] () ::=
    Channel >-> verifyLiveness [ Channel ] ()
    -- Ensure the sequence keeps progressing.
ENDDEF

MODELDEF SpecVerifyLiveness ::=
    CHAN IN
    CHAN OUT   Channel1
    BEHAVIOUR
        verifyLiveness [ Channel1 ] ()
ENDDEF

-- ------------------------------------------------------------
-- 2. Deadlock Freedom: Ensure that the system does not deadlock.
-- ------------------------------------------------------------
-- This test verifies that the system cannot get stuck
in a deadlock situation.
PROCDEF verifyNoDeadlock [ Channel1, Channel2 ] () ::=
    ( Channel1 >-> EXIT ) -- Simple behavior for Channel1
    |||
    ( Channel2 >-> EXIT ) -- Simple behavior for Channel2
ENDDEF

MODELDEF SpecVerifyNoDeadlock ::=
    CHAN IN
    CHAN OUT   Channel1, Channel2
    BEHAVIOUR
        verifyNoDeadlock [ Channel1, Channel2 ] ()
        -- Ensure no deadlock occurs.
ENDDEF

-- ------------------------------------------------------------
-- 3. Synchronization Verification: Ensure processes synchronize
correctly.
```

```
-- ----------------------------------------------------------------
-- This test ensures that two processes synchronize correctly on
shared channels.
PROCDEF verifySynchronization [ Channel1, Channel2 ] () ::=
    Channel1 >-> ( Channel2 >-> verifySynchronization [ Channel1,
    Channel2 ] () )
    |[ Channel1, Channel2 ]|
    ( Channel2 >-> Channel1 >-> verifySynchronization [ Channel1,
    Channel2 ] () )
ENDDEF

MODELDEF SpecVerifySynchronization ::=
    CHAN IN
    CHAN OUT   Channel1, Channel2
    BEHAVIOUR
        verifySynchronization [ Channel1, Channel2 ] ()
        -- Verify synchronization between two channels.
ENDDEF


-- ----------------------------------------------------------------
-- 4. Data Integrity Verification: Ensure valid data flows through
channels.
-- ----------------------------------------------------------------
-- This test verifies that data sent through channels follows the
specified rules.
PROCDEF verifyDataIntegrity [ Channel :: Int ] () ::=
    Channel ? x  >-> verifyDataIntegrity [ Channel ] ()
    -- Ensure that the data follows the integrity constraints.

ENDDEF

MODELDEF SpecVerifyDataIntegrity ::=
    CHAN IN
    CHAN OUT   ChannelInt1
    BEHAVIOUR
        verifyDataIntegrity [ ChannelInt1 ] ()
        -- Verify that data follows the integrity rules.
ENDDEF

-- ----------------------------------------------------------------
-- 5. Correct Handling of Choices: Verify choice behavior.
```

```
-- ------------------------------------------------------------
-- This test ensures that the model correctly handles multiple choices.
PROCDEF verifyChoiceHandling [ Channel1, Channel2 ] () ::=
    Channel1 >-> verifyChoiceHandling [ Channel1, Channel2 ] ()
    ##
    Channel2 >-> verifyChoiceHandling [ Channel1, Channel2 ] ()
ENDDEF

MODELDEF SpecVerifyChoiceHandling ::=
    CHAN IN
    CHAN OUT   Channel1, Channel2
    BEHAVIOUR
        verifyChoiceHandling [ Channel1, Channel2 ] ()
        -- Verify that choices are handled correctly.
ENDDEF

-- ------------------------------------------------------------
-- 6. Concurrency Verification: Ensure concurrent processes operate
correctly.
-- ------------------------------------------------------------
-- This test verifies that multiple processes can run concurrently
without interference.
PROCDEF verifyConcurrency [ Channel1, Channel2 ] () ::=
    Channel1 >-> EXIT
    |||
    Channel2 >-> EXIT
ENDDEF

MODELDEF SpecVerifyConcurrency ::=
    CHAN IN
    CHAN OUT   Channel1, Channel2
    BEHAVIOUR
        verifyConcurrency [ Channel1, Channel2 ] ()
        -- Verify correct operation of concurrent processes.
ENDDEF

-- ------------------------------------------------------------
-- 7. Extended Synchronization Test: Verifies synchronization of
multiple processes.
-- ------------------------------------------------------------
-- This test ensures that multiple processes can synchronize
on shared channels.
```

```
PROCDEF verifyMultiProcessSync [ Channel1, Channel2, Channel3 ] () ::=
    Channel1 >-> ( Channel2 >-> Channel3 >-> EXIT )
    |[ Channel1, Channel2, Channel3 ]|
    Channel2 >-> ( Channel3 >-> Channel1 >-> EXIT )
ENDDEF

MODELDEF SpecVerifyMultiProcessSync ::=
    CHAN IN
    CHAN OUT   Channel1, Channel2, Channel3
    BEHAVIOUR
        verifyMultiProcessSync [ Channel1, Channel2, Channel3 ] () -
        - Verify multi-process synchronization.
ENDDEF

-- ------------------------------------------------------------
-- 8. Nested Synchronization Verification: Ensure correct nested
synchronization.
-- ------------------------------------------------------------
-- This test ensures that nested synchronized processes behave as
expected.
PROCDEF verifyNestedSync [ Channel1, Channel2 ] () ::=
    ( Channel1 >-> EXIT )
    |[ Channel1 ]|
    ( Channel2 >-> EXIT )
ENDDEF

MODELDEF SpecVerifyNestedSync ::=
    CHAN IN
    CHAN OUT   Channel1, Channel2
    BEHAVIOUR
        verifyNestedSync [ Channel1, Channel2 ] ()
        -- Verify nested synchronization between two channels.
ENDDEF

-- ------------------------------------------------------------
-- Deadlock Inducing Test
-- ------------------------------------------------------------

PROCDEF induceDeadlock [ Channel1, Channel2 ] () ::=
-- Channel1 waits for an event but will never receive it,
as Channel2 is blocked
    ( Channel1 >-> EXIT )
```

```
    |||
    -- Channel2 waits for an event but will never receive it,
    as Channel1 is blocked
    ( Channel2 >-> Channel1 >-> EXIT )
ENDDEF

MODELDEF SpecInduceDeadlock ::=
    CHAN IN
    CHAN OUT   Channel1, Channel2
    BEHAVIOUR
        induceDeadlock [ Channel1, Channel2 ] ()
        -- This will induce a deadlock.
ENDDEF
```

A.7 Integration Test

The Integration Test uses several custom MODELDEF and PROCDEF constructs to simulate real-world scenarios where components interact in potentially error-prone ways. Each aspect of the test is designed to address specific integration challenges.

Deadlock Scenario: A synchronized process (synchronizedN) expects three signals to synchronize but is intentionally provided with only one active process. This creates a potential deadlock due to unfulfilled synchronization requirements. If a process waits indefinitely for a message on 'Channel1', but that message never arrives due to a synchronization issue with 'synchronizedN'. For example, if 'synchronizedN' expects three processes to synchronize, but only two are active, it could cause the system to deadlock, as it waits for a third signal that will never come. During execution one synchronizing process is disabled to observe whether the system deadlocks as it waits indefinitely for missing signals.

Race Condition Scenario: Suppose both 'Channel2' and 'Channel3' are used in a 'choice' process, where the path depends on which channel receives data first. If multiple instances of 'parallelN' interact with these channels without enforced order, it could lead to unpredictable outcomes, especially if 'parallelN' executes more quickly than anticipated.

During execution we increase the number of parallel processes and observe interleaving of actions on shared channels to identify non-deterministic outcomes For example, we can increase the number of parallel processes interacting with 'Channel2' and 'Channel3' in a way that would create contention and observe if outcomes vary across test runs, indicating non-deterministic behavior.

Incorrect Message Passing: Setup: If 'Channel4' and 'Channel5' expect specific data types but receive different ones due to mismatched message formats between processes, the test could detect this fault. For example, if 'sequence' on 'Channel1' expects integers but 'choice' on 'Channel2' sends strings, the process will not be able to proceed.

'Channel4' is defined to expect integers ('Int') while 'Channel5' is defined to expect booleans ('Bool'). 'processF' sends an integer on 'Channel4', matching the expected type. 'processG' sends a boolean on 'Channel5', also matching the expected type. In the meantime 'incorrectProcess' tries to send a string ('"incorrectmessage"') on 'Channel4', which expects an integer. This deliberate mismatch should trigger a type error. Similarly, 'incorrectProcess2' tries to send an integer ('0') on 'Channel5', which expects a boolean. This is another deliberate mismatch. We then deliberately configure one process to send an incompatible data type through 'Channel4' or 'Channel5' to see if the test identifies the mismatch.

To simulate an Incorrect Message Passing scenario, we can deliberately set up mismatched data types for 'Channel4' and 'Channel5'. In this example, we'll configure 'Channel4' to expect integers, but one of the processes will attempt to send a string on this channel, creating a data type mismatch. Similarly, for 'Channel5', we'll set up an expected data type and introduce a mismatch.

Branching Errors: A choice process responds to inputs on Channel2 and Channel3. Valid inputs transition to successBranch, while invalid inputs transition to errorBranch. If the 'choice' process does not respond correctly to inputs from 'Channel2' or 'Channel3', it might take the wrong path or fail to respond, leading to a branching error. For example, if 'choice' is meant to respond only to certain signals but incorrectly takes an alternative path, it would indicate a fault.

During execution we set up test cases where 'choice' receives inputs that should lead to each possible branch, verifying that it responds correctly each time. Introducing unexpected inputs can also reveal if the process mishandles such cases. We then send valid and invalid inputs on Channel2 and Channel3 in parallel and observe transitions to successBranch or errorBranch.

Here is the complete model definition:

```
MODELDEF IntegrationTestDeadlockScenario ::=
    CHAN IN    Channel1, Channel2, Channel3
    CHAN OUT   Channel4, Channel5
    BEHAVIOUR
        (
            -- Test the integration of sequence with choice
            sequence [ Channel1 ] ()
        |||
            choice [ Channel2, Channel3 ] ()
        |||
-- Synchronized process expecting three signals but now only one
process will participate in synchronization
            synchronizedN [ Channel1 ] (3)
        )
        |[ Channel1, Channel2, Channel3 ]|
        (
            sequence [ Channel1 ] ()
        )
        |[ Channel1, Channel4, Channel5 ]|
        (
-- Only one process is now synchronizing on Channel1, instead of
the expected three
            parallelN [ Channel4 ] (2)
```

```
        )
ENDDEF

MODELDEF ComplexRaceConditionTest ::=
    CHAN IN    Channel1, Channel2, Channel3, Channel4, Channel5
    CHAN OUT
    BEHAVIOUR
        (
            processA [ Channel1 ] ()
        |||
            processB [ Channel2 ] ()
        |||
            processC [ Channel3 ] ()
        |||
            processD [ Channel4 ] ()
        |||
            processE [ Channel5 ] ()
        )
ENDDEF

-- Process A: Continuously activates Channel1
PROCDEF processA [ Channel1 ] () ::=
    Channel1 >-> processA [ Channel1 ] ()
ENDDEF

-- Process B: Continuously activates Channel2
PROCDEF processB [ Channel2 ] () ::=
    Channel2 >-> processB [ Channel2 ] ()
ENDDEF

-- Process C: Continuously activates Channel3
PROCDEF processC [ Channel3 ] () ::=
    Channel3 >-> processC [ Channel3 ] ()
ENDDEF

-- Process D: Continuously activates Channel4
PROCDEF processD [ Channel4 ] () ::=
    Channel4 >-> processD [ Channel4 ] ()
ENDDEF

-- Process E: Continuously activates Channel5
PROCDEF processE [ Channel5 ] () ::=
    Channel5 >-> processE [ Channel5 ] ()
```

```
ENDDEF

CHANDEF Channels ::=
    Channel1 :: Int;
    Channel2 :: Int;
    Channel3 :: Int;
    Channel4 :: Int;
    Channel5 :: Bool;
ENDDEF

MODELDEF IncorrectMessagePassingTest ::=
    CHAN IN    Channel1, Channel2, Channel3, Channel4, Channel5
    CHAN OUT
    BEHAVIOUR
        (
            -- Process A sending an integer on Channel4 as expected
            processF [ Channel4 ] ()
        |||
            -- Process B trying to send a string on Channel4,
            causing a type mismatch
            incorrectProcess [ Channel4 ] ()
        |||
            -- Process C sending a boolean on Channel5 as expected
            processG [ Channel5 ] ()
        |||
            -- Process D trying to send an integer on Channel5,
            causing a type mismatch
            incorrectProcess2 [ Channel5 ] ()
        )
ENDDEF

-- Process F: Correctly sends an integer on Channel4
PROCDEF processF [ Channel4 ] () ::=
    Channel4 ! (1) >-> processF [ Channel4 ] ()
ENDDEF

-- Incorrect Process B: Tries to send a string on Channel4,
causing a type mismatch
PROCDEF incorrectProcess [ Channel4 ] () ::=
    Channel4 ! ("incorrect_message") >-> incorrectProcess [ Channel4 ]()
ENDDEF

-- Process G: Correctly sends a boolean on Channel5
```

```
PROCDEF processG [ Channel5 ] () ::=
    Channel5 ! (True) >-> processG [ Channel5 ] ()
ENDDEF

-- Incorrect Process D: Tries to send an integer on Channel5,
causing a type mismatch
PROCDEF incorrectProcess2 [ Channel5 ] () ::=
    Channel5 ! (0) >-> incorrectProcess2 [ Channel5 ] ()
ENDDEF

CHANDEF Channels ::=
    Channel1 :: Int;  -- Output channel
    Channel2 :: Int;  -- Input channel
    Channel3 :: Int;  -- Input channel
ENDDEF

MODELDEF BranchingErrorTest ::=
    CHAN IN    Channel2, Channel3
    CHAN OUT   Channel1
    BEHAVIOUR
        (
            choiceProcess [ Channel1, Channel2, Channel3 ] ()
        |||
            sendExpectedValueChannel2 [ Channel2 ] ()
        |||
            sendUnexpectedValueChannel3 [ Channel3 ] ()
        )
ENDDEF

PROCDEF choiceProcess [ Channel1, Channel2, Channel3 :: Int]():::=
    (
Channel2 ? x [[ x == 5 ]] >-> successBranch [ Channel1, Channel2,
Channel3 ] ()
    ##
Channel3 ? y [[ y == 10 ]] >-> successBranch [ Channel1, Channel2,
Channel3 ] ()
    ##
Channel2 ? x >-> errorBranch [ Channel1 ] ()
    ##
Channel3 ? y >-> errorBranch [ Channel1 ] ()
    )
ENDDEF
```

```
PROCDEF successBranch [ Channel1, Channel2, Channel3 :: Int ]():=
    Channel1 ! (1) >-> choiceProcess [ Channel1, Channel2,
    Channel3 ] ()
ENDDEF

PROCDEF errorBranch [ Channel1 :: Int  ] () ::=
    Channel1 ! (99) >-> errorBranch [ Channel1] ()
ENDDEF

PROCDEF sendExpectedValueChannel2 [ Channel2 :: Int ] () ::=
    Channel2 ! (5) >-> sendExpectedValueChannel2 [ Channel2 ] ()
ENDDEF

PROCDEF sendUnexpectedValueChannel3 [ Channel3 :: Int  ] () ::=
    Channel3 ! (99) >-> sendUnexpectedValueChannel3 [ Channel3 ] ()
ENDDEF
```