Formal Analysis of DDML

A framework for studying DDML Models

Soremekun Olamide Ezekiel (40151)

Abstract

We propose a framework for formal analysis of DEVS Driven Modeling Language (DDML) models in order to assess and evaluate the properties of DDML models. This framework semantically maps the hierarchical levels of DDML: Input Output system (IOS), Input Output Relation Observation (IORO) and Coupled Network (CN) levels to corresponding formal methods: Labeled Transition System (LTS), Linear Temporal Logic (LTL) and Computation Tree Logic (CTL), and Communicating Sequential Processes (CSP) respectively. These formal methods capture the semantics of DDML at each level of abstraction and we use formal tools (such as JTORX, LTSA, PAT and NUSMV) to automatically analyze these formal specifications to evaluate properties of DDML models at each level.

Table of Content

Contents

Chapter 1: Introduction

The framework proposed in this work is intended to formally assess and evaluate DDML models. DDML is a modeling language designed to be used for the DEVS formalism of modeling and simulation (M & S). Modeling and simulation is a broad field of science applicable to computer science through computer automated and monitored simulation for real life problems in science and engineering. The applications of computer simulations cover a broad spectrum of fields and are relevant to study real life systems in order to gather information and make strategic decisions. In order to delve into the bulk of this work, it is important to give an overview of the M & S field and reveal its importance, uses and concepts.

1.1 Modeling and simulation

Modeling and simulation is a paradigm that provides a way of representing problems and reasoning about the problems in order to proffer a solution or method or absorb relevant data to solve such problems.

Concepts

There are five major concepts of Modeling and simulation that help define the process and the components of the process. Below is the definition of such concepts:

- An Object: this is a certain entity that exists in the real world which has a specific structure and behavior.
- A system: is a well defined object in the real world under specific conditions, which define specific aspects of the object's structure and behavior.
- A model: is an abstract representation of an object, its properties, structure and behavior. This representation could be logical, mathematical or physical.
- Experimental Frame: the experimental frame defines the experimental conditions under which the model would be used. It encapsulates the objectives of the experiment and the aspects of the model to be tracked.
- Simulation: is the execution of a model, in order to get information about the changes in the behavior, structure and properties of the system during the executions. It helps to observe and infer on the dynamic behavior of the system by showcasing the implementation of these dynamic behaviors over time.

Modeling and simulation refers to the process of producing a model of a real life system and the act of operating on the model in order to study and absorb new information. Modeling and simulation in computer science is used to study the properties, structure ad behavior of systems and to gather and infer new information concerning such systems. Modeling and simulation has become widely used in many aspects of science and even in business in order to study events and activities through models likened to the real world scenarios. This is because virtual experimentation of models is cheaper, possible and even safer to study for critical systems such as: aviation, health care, drug production, electrical/electronic systems, oil production, supply chain management and logistics, military intelligence and defense systems. However, in computer science modeling and simulation can be used

at many stages of development from the requirements engineering phase to the design, implementation, project management and the installation phase. Below is a diagram showing the use of Modeling and simulation within the computer science field.

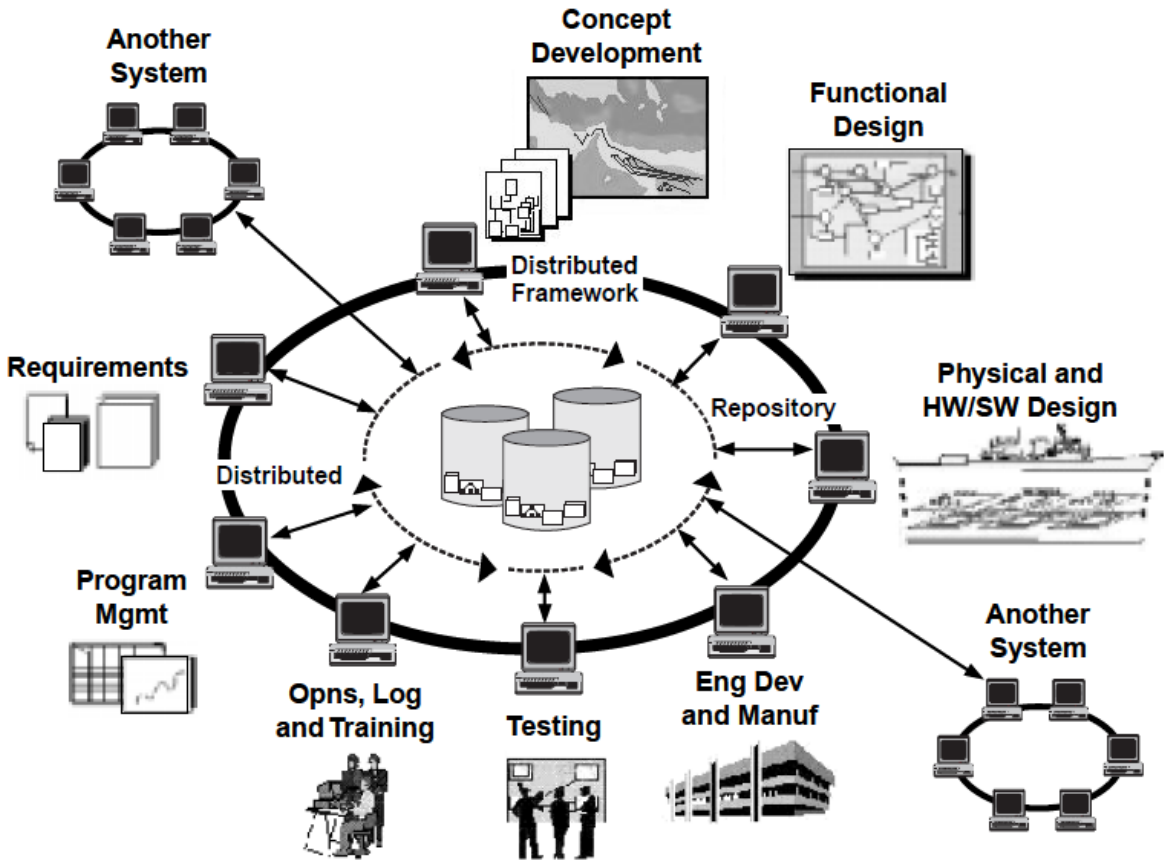


Fig 1: the use of Modeling and Simulation in the Computer Science field

Benefits

Modeling and simulation can be of many benefits to the user and the world at large. Modeling and simulation helps the user to understand the system better, optimize the performance of the system and check for reliability and safety of such systems. With modeling and simulation, different constraints and conditions that can not be checked in the real world can be checked by the user. For example life critical scenarios in the health care system or drug use or even microscopic level of detail evaluation of material can be performed by simulation of their models. Furthermore, modeling and simulation has been a tool to help validate systems, models of existing systems are executed and errors or bugs are checked in order to avoid their disastrous occurrence in real life. For example, models of the road and rail transport system in Europe as well as the air traffic control system in the United States have been tested and validated through modeling and simulation [1, 2].

Furthermore, modeling and simulation is essential for the effective design and evaluation of systems, it helps to build a reliable and well designed product or system. This is achieved by studying the operation of the proposed system through its model over time and improving its performance by manipulating constraints and conditions that affect the system in order to compute the best set of constraint variable for the best performance. This way the effects of changes in conditions on the system can be determined without actually affecting the real system or having to build the system to check for such changes. The diagram shown below in figure 2 help summarize the numerous advantages of Modeling and simulation in our society.

Finally, modeling and simulation provides a way to solve real life problems or even solve unforeseen or future resulting faults in systems. Through simulation, the time effect of long processes can be compressed in order to see the possible future outcome of processes and faults that can be generated in the course of future operation. Identifying such faults can help make important business and government decision concerning the system in order to avert faults or possible future disaster. Besides this, a real life problem like the spread of a particular disease can be modeled and the spread can be cut off by noting the rate or direction of spread through the simulation and applying necessary decisive strategies to avert this.

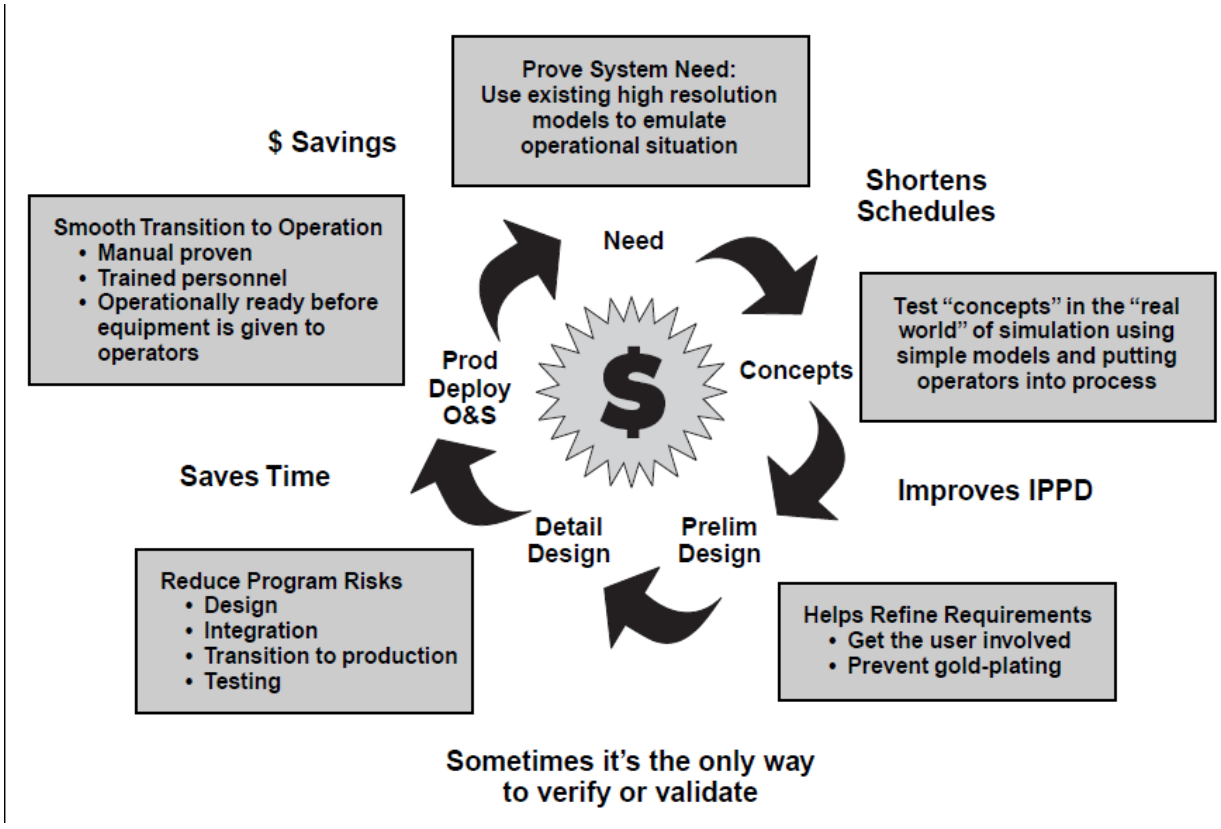


Fig 2: the numerous advantages of Modeling and Simulation

Importance

The importance of modeling and simulation is pinnacled on its strengths. It is important because it helps in saving money in systems testing. Modeling and simulations is cheaper than building the system and performing test cases on the built system, this would involve building many duplicate systems has systems can be destroyed after each test case. However, with a good model of the system, test cases can be performed without the need to spend huge expenses building the same system over and over again.

Besides this, Modeling and Simulation (M & S) provides a level of detail that is flexible to system size. A model can represent a system as big as the universe or as small as a microscopic environment with an efficient level of detail that is important to the experimental frame. This level of detail is important because it might not be achievable in the real system with the current technology but can be achieved through models and notable information about the behavior of such real life systems can even be inferred from such model. This flexibility in size is a major strength of modeling and simulation. It provides a cutting edge advantage for this form of system experimentation.

Challenges

Every paradigm has its limitations and challenges. Modeling and computer simulation can be limited by the capacity of computers in terms of memory, processing speeds or other resources. Another major limitation could be the amount of data available to accurately represent the system. A major challenge of M & S is the ability to accurately and efficiently represent the real life system with the best properties, structure and behavior that correctly depict the system. A shortage of this accurate representation may result in a model error and consequently simulation errors.

Another M & S challenge is the use of an incorrect simulation algorithm or program; this would result in simulation errors or inaccurate behavior. Simulation programs that are incorrect could provide its users with wrong information on the behavior, properties and structure of the system. Subsequently, this would make the user infer wrong data about the system and creates a faulty simulation result. This would result in a process with a questionable credibility.

Finally, it could be difficult to correctly interpret simulation results and to also infer important or accurate information form the simulation results. Verifying and validating the simulation results could prove abortive, and the understanding of some results in the real life might be difficult to attain. The strength of the modeling and simulation process lies in the correct interpretation of its result, challenges interpreting these results could make the entire process useless.

1.2 Introduction to DDML and DEVS

The Discrete Event System Specification (DEVS) is a modeling formalism that has become important and highly preferable in modeling and simulation; this is because other formalisms have been proven to have an equivalent DEVS representation. Even though DEVS is specifically designed to model discrete

events, it has been shown that discrete time events and differential (continuous) event systems can be easily converted to DEVS. DEVS is also important because of its modular architecture which promotes the separation of components/concerns (model, simulator and experimental frame). Furthermore, DEVs is semi formal which allows for freedom and flexibility of use, modelers are free to express the structure, behavior and traces of their system freely.

DDML (DEVS modeling language) is a graphical and hierarchical formalism that is developed to construct models of dynamic systems for simulation. DDML is motivated by DEVS and it is developed to capture the structure and behavior of systems by focusing on the three levels of abstraction, the Input Output System (IOS) level, the Coupled Network (CN) level and the Input Output Relation Observation (IORO) level.

1.3 Formal methods

A formal specification is the expression, in some formal language and at some level of abstraction, of a collection of properties some system should satisfy. It is important to view the idea of specifications pertaining to the problem domain (as opposed to the solution domain). To make sure some solution solves a problem correctly, one must first state that problem correctly.

A formal specification language provides a formal method's mathematical basis. A specification is formal if it is expressed in a language made of three components: rules for determining the grammatical well-formedness of sentences (the syntax); rules for interpreting sentences in a precise, meaningful way within the domain considered (the semantics); and rules for inferring useful information from the specification (the proof theory).

Formal specification techniques essentially differ from semi-formal ones (such as dataflow diagrams, entity-relationship diagrams or state transition diagrams) in that the latter do not formalize the assertion part: an assertion part, where the intended properties on the declared variables are formalized.

A formal specification language provides a notation (its syntactic domain), a universe of objects (its semantic domain), and a precise rule defining which objects satisfy each specification. A specification is a sentence written in terms of the elements of the syntactic domain

1.4 Structure of thesis report

The rest of this report is structured as follows. Chapters 1, 2 and 4 provide an introductory section to the work done. In chapter 2 we provide a brief introduction to DDML, its syntax and semantics. Chapter 3 provides a look into the field of formal methods, a survey of its concepts and paradigms, its benefits and a summary of formal tools used in the course of this work.

The next set of chapters builds on the introductory topics and provides a basis for the work done. Considering the introductory topics touched in the first three chapters, in chapter 4 we link these concepts by discussing the importance, approaches and challenges involved in combining formal methods and simulation. Furthermore, this chapter reveals the importance of using multiple formal methods in the course of our work. In chapter 5, we propose a formal framework for DDML and rationalize our choice of formal methods at each level of abstraction and provide a semantic mapping/translation of each level of DDML abstraction to each formal method.

The bulk of the work is shown in chapter 6 by revealing the formal analysis of DDML models at all three levels of abstraction through a traffic light case study. Finally, chapter 7 provides a summary of the work, the challenges involved, the need to integrate the tools and a peep into the future work that would be continued on this work. In this chapter we reveal our intention to provide a tool for automatic code generation for formal analysis and propose architecture for this platform.

Chapter 2: The DEVS-Driven Modeling Language

DEVS Driven Modeling Language (DDML) provides a graphical notation for DEVS. It is a comprehensive and intuitive way of expressing DEVS through a visual language. It provides a means for users to reason and study systems. DDML focuses on three hierarchical levels of abstraction according to the Ziegler levels of knowledge abstraction, these three levels are namely [6]:

- Input Output System (IOS) concerned with system dynamic behaviors, especially the change from one state to another and the activities or input stimuli causing such changes as well as the output of the system after each input stimuli. This level is characterized by states and state transitions.
- Input Output Relation Observation (IORO) is concerned with traces and trajectories of the system i.e. the footprints of the system dynamic behaviors.
- Coupled Network (CN) is concerned with the structural properties and functional couplings of the system components.

Formally, the DDML logical semantics are mapped into these three levels of abstraction in order to capture the relevant information important at each level. In order to asses system properties at each level, different formal methods adaptive to the specialization of each level are employed to help formally express the DDML framework. In view of this, the different tools that correspond to each level of abstraction and semantic definition would also be employed at each level.

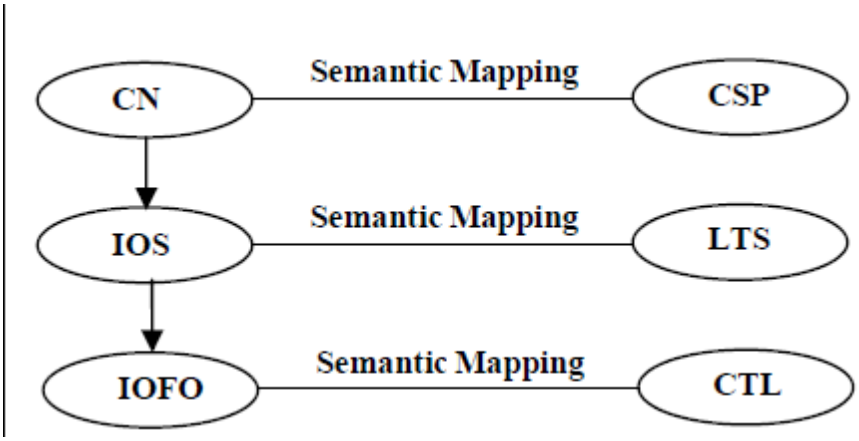


Fig 3: Semantic mapping of DDML hierarchical levels to corresponding formal methods

This hierarchical system specification at each level is intended to capture the important information of the DDML. At the IOS level, the DDML semantics are mapped to the labeled transition system (LTS) formal method in order to capture the transition based formal definition of the model. At the IORO level, the DDML semantics are mapped to the Linear Temporal Logic (LTL) [7] and the Computation Tree Logic (CTL) [7] semantics in order to capture the temporal logic based formal definition of the modeling language. Finally, the CN level of DDML is mapped to the Communicating Sequential Process (CSP) [7] formal method in order to capture the process algebraic form of the coupled model and the interaction of its component parts.

DDML: Syntax

Abstract syntax

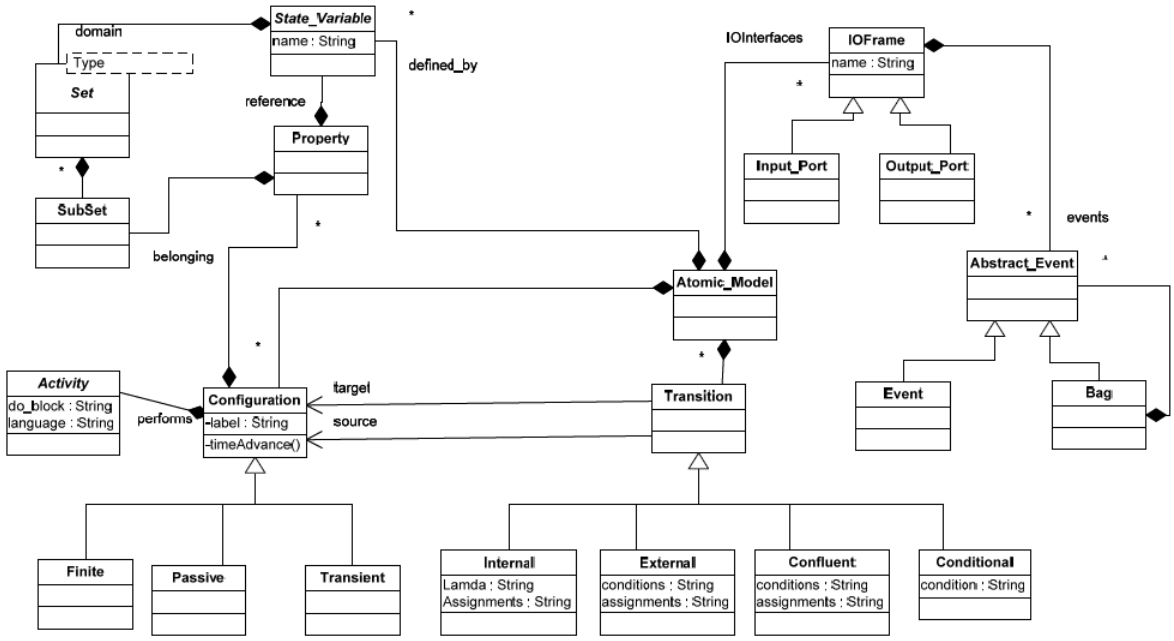


Figure 4: DDML IOS Meta-model. The following constraints apply: for Transient state: ta = 0; Passive State: ta = +∞; Finite state 0 < ta < + ∞

The figure 4 above shows the abstract syntax of the DDML IOS meta-model. The abstract model captures the "AtomicModel" by representing the components of the system. Each component can be at a state or another, these states are described by "StateVariables". The interactions and relationships between components and the environment is modeled by the "AbstractEvents", the channels through these interactions are performed is called the "IOInterfaces" representing the "InputPorts" or "OutputPorts". Events that occur concurrently in Parallel DEVS [7] are called Bags of events. A system component may have many states and its state space can be represented by an infinite graph of states which may lead to the state explosion problem. In order to curb this problem, state clusters are represented by a finite number through "Configurations".

A "Configuration" is a partition or subdivision of the state space into non-overlapping and nonempty subsets of states. As such, a configuration has the following properties [9]:

- Every state in the state space belongs to a particular *Configuration*
- The sets of Configurations are mutually disjoint

- The states in a “Configuration” are “alike” (as a result of the configuration on state variables and similar transitions to states belonging to the same *Configuration*); thus they are related by an equivalence relation and a “Configuration” is an equivalence class of states
- A “Configuration” is defined by *Properties* and a *Property* is a cross product of *sub sets* of the domains of the state variables*).*

Concrete syntax

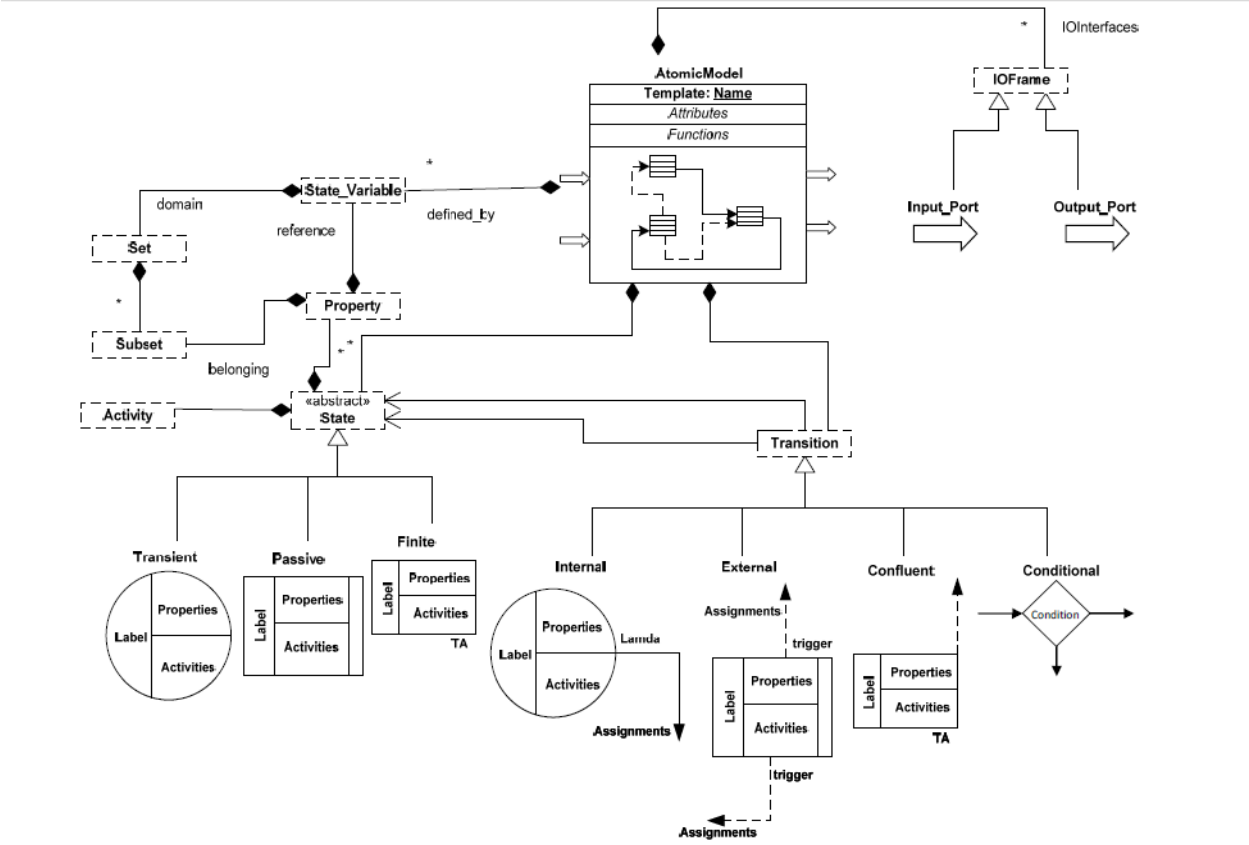


Fig 5: DDML IOS Concrete Syntax. Some attributes do not have graphical notations and are inserted into other graphical elements.

The DDML “AtomicModel” like other visual languages (like OOML and UML) can be used to capture some system properties like inheritance, aggregation and composition. Stimuli sent through input ports are shown above and these ports are nomenclature with their name and type in this format- (name: type), likewise the output ports. The atomic model is described with the name, attributes and functions and the last compartment as shown above in the graphical notation illustrates the state chart of the component. The state chart expresses four functions graphically (corresponding to *δint, δext, λ,* and *ta*

in DEVS) and the notation for *Configuration* is a box with four compartments for label, properties, activities, and time advance [9]. External transitions are represented with broken arrows which appear before the edge of the box and directed towards the lateral sides, while internal transitions are represented with a straight line at the right edge of the box directed towards the left edge of another box. Confluent transition originates from the top right corner and is directed towards the back. To illustrate we shall consider a DDML IOS model of the Eiffel Tower (Fig. 6).

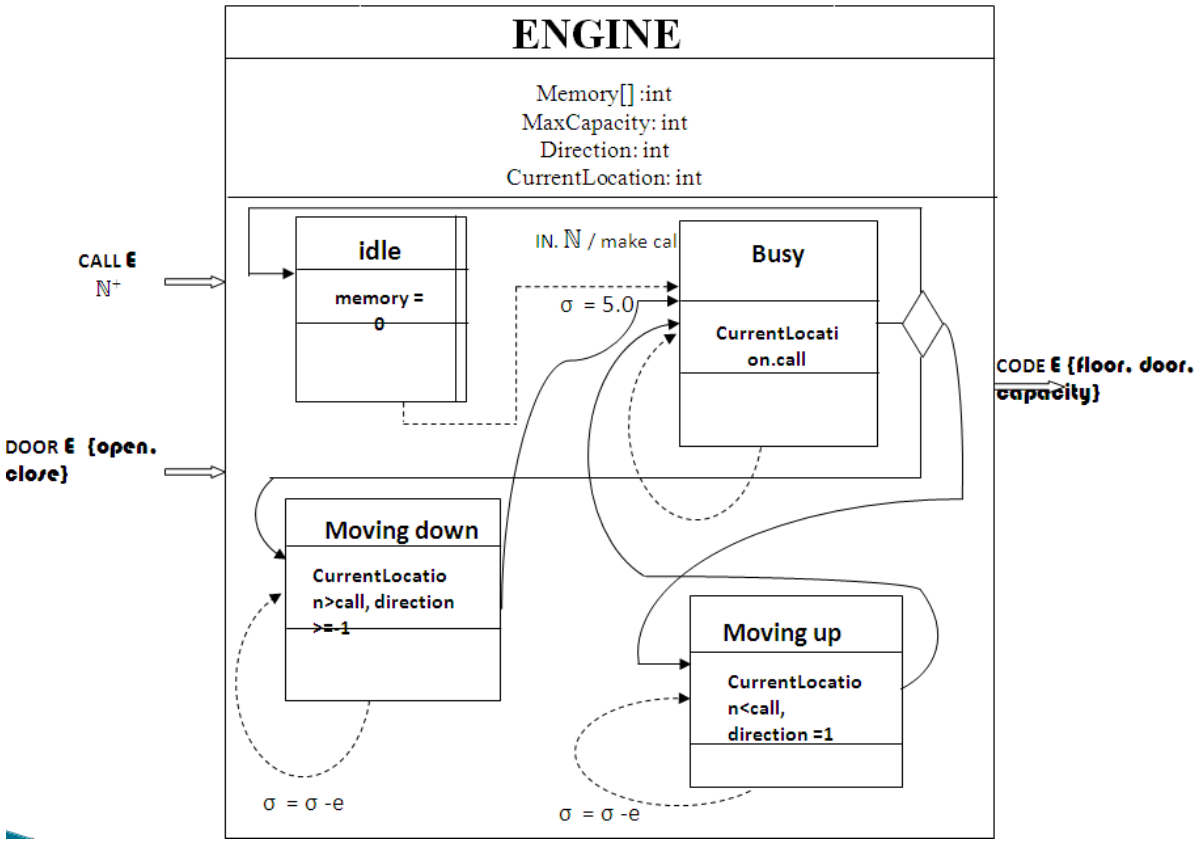


Figure 6: Example of DDML IOS model of Eiffel Tower

The model above represents the IOS level of the DDML Eiffel Tower system. This model shows that the Eiffel Tower engine can be in four different states at different times in its life cycle. Furthermore, the model reveals the important state variables that determine and affect the different states at each time. These state variables as shown in the second rectangle below the IOS Model name “Engine” includes: the Eiffel tower engine memory (Memory[]), the Maximum capacity of the cabin (MaxCapacity), the direction of the Cabin movement (Direction) and the current location of the cabin (CurrentLocation).

The Eiffel Tower is initially in the “Idle” state, in which its memory is empty (memory =0). When a call is made in the cabin for a floor, the state of the Eiffel Tower engine becomes “Busy”. From the “Busy” state the Eiffel tower Engine state changes to “Moving down” if the current location of the cabin is greater than the floor call made and it changes to “Moving up” if the current location of the cabin is lower than the floor call made. From the “Moving up” and the “Moving down” states the system can go

to the "Busy" state if a new cabin floor call is made while moving or remain in this state till the cabin floor call made is reached.

Furthermore, to illustrate the DDML concrete syntax, the coupled model of the lift system of the Eiffel Tower is shown below in Fig 7. This model shows the interaction among the Eiffel tower system components. It shows that the input into the system includes the Door opening or closing ("DOOR") and the floor call ("CALL"). Through processes in the engine component has explained earlier, messages determining the system movement are sent to the cabin and the Eiffel tower movements are determined.

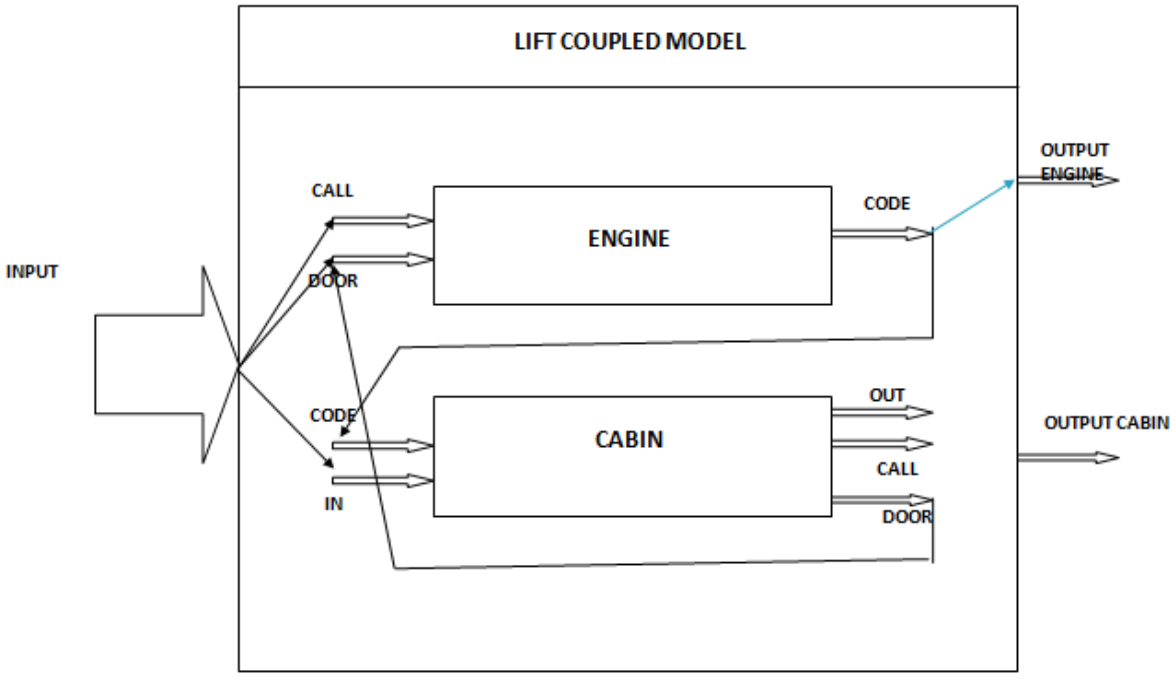


Figure 7: Example of DDML CN model of Eiffel Tower

DDML: Semantics

Formally, a DDML Atomic Model is a tuple of the form [8]:

B = <XB, YB, SB, SOB, ψ , C(VB), Tint, Text, C(e), Op, φ, Act >

- XB: set of input ports
- YB: set of output ports
- SB: finite set of state clusters
- SOB: initial state of B
- Ψ: SB → $\mathbb{P}\ C\ (\mathrm{VB})$: is a mapping between each element of SB and a finite set of conditions on the variable in VB i.e. a mapping between the states and the state variable conditions

- C (VB): set of Constraints on state variables
- Tint: is the set of internal transitions
- Text: is the set of external transitions
- C (e): is the set of conditions α of the elapsed time e
- Op: is the set of operations defined on the state variables
- φ: SB → $\mathbb{P}\ Act$: is the mapping from states to the set of activities
- Act: is the set of activities

Formally a DDML IORO is a tuple of the form <T, X, Ω, Y, R >, where:

- <T, X, Y> is the observation frame (T-Time, X-Input, Y-Output)
- Ω is the set of all possible input segments
- Ω ⊆ (X, T)
- R is the I/O relation
- R ⊆ Ω X (Y, T)
- $(\omega, \rho) \epsilon$ R ⇒ dom (ω) = dom (ρ)
- ω: <t_i, t_f> → X: input segment
- ρ: <t_i, t_f> → Y: output segment

Formally a DDML CN level, for each component of the DDML CN a type definition for their names is required; this is shown below [8]:

$$[ModelName, PortName, CouplingName]$$

A port has a name and a direction (in or out corresponding to input or output port types respectively).

$$Direction ::= OUT \mid IN$$

Chapter 3: Formal Methods

3.1 Introduction to formal methods

A formal specification is the expression, in some formal language and at some level of abstraction, of a collection of properties some system should satisfy. A formal specification language provides a formal method's mathematical basis. A specification is formal if it is expressed in a language made of three components: rules for determining the grammatical well-formedness of sentences (the syntax); rules for interpreting sentences in a precise, meaningful way within the domain considered (the semantics); and rules for inferring useful information from the specification (the proof theory).

Formal specification techniques essentially differ from semi-formal ones (such as dataflow diagrams, entity-relationship diagrams or state transition diagrams) in that the latter do not formalize the assertion part: an assertion part, where the intended properties on the declared variables are formalized.

A formal specification language provides a notation (its syntactic domain), a universe of objects (its semantic domain), and a precise rule defining which objects satisfy each specification. A specification is a sentence written in terms of the elements of the syntactic domain

Syntactic domain: a set of symbols (for example, constants, variables, and logical connectives) and a set of grammatical rules for combining these symbols into well-formed sentences.

Semantic domain: Concurrent and distributed systems specification languages are used to specify state sequences, event sequences, state and transition sequences, streams, synchronization trees, partial orders, and state machines.

Satisfies domain: Two broad classes of semantic abstraction functions are those that abstract preserving each system's behavior and those that abstract preserving each system's structure

3.2 Benefits of formal methods

The benefits of formal methods in software development include:

1. Formal specifications help expose ambiguities, inconsistencies and incompleteness in the requirements of a system.
2. Formality helps in obtaining higher-quality specifications within processes (designing, validating, documenting, communicating, reengineering, and reusing solutions); it also provides the basis for their automated support.
3. The semantics of the formalism being used provides precise rules of interpretation that allow many of the problems with natural language to be overcome.

4. It helps in the analysis of a system in respect to its requirements, without the necessity of prototyping or repeated test cases on the system itself.

Uses of Formal Methods:

Requirements analysis: can be used to specify the functional requirements of the proposed system. It is essential for documenting the user requirements and brainstorming about the problem domain.

System design: A specification language can help in designing the system architecture, its modules or its functions.

System verification: Verification is the process of showing that a system satisfies its specification.

System validation: Formal methods can aid in system testing and debugging. Specifications that explicitly state assumptions on a module's use identify test cases for boundary conditions.

System documentation: A specification is a description alternative to system implementation. It serves as a communication medium between a client and a specifier, between a specifier and an implementer, and among members of an implementation team.

System analysis and evaluation: To learn from the experience of building a system, developers should do a critical analysis of its functionality and performance once it has been built and tested.

In this project, formal specifications and its automated tools were used for the following purposes:

- to generate concrete scenarios illustrating desired or undesired features about the DDML
- to check specific forms of specification consistency/completeness efficiently of DDML
- to generate high-level exceptions and conflict preconditions that may make the DDML specification not satisfiable
- to generate higher-level specifications such as invariants or conditions for liveness
- to drive refinements of the specification and generate proof obligations
- to generate test cases and oracles from the specification
- to check DDML for adequacy and consistency with the models requirements

3.3. Survey of formal methods

A formal specification method could be based on logic, state machines or process algebra. Logic based formal methods include Z specification, First order logic and temporal Logic. State machines include: finite state machines, state charts, communicating state machines and automata. Lastly, Process algebra

includes Calculus of Communicating Systems (CCS), Communicating Sequential Processes (CSP) and Process-Algebraic Analysis of Real-Time Applications (PARAGON).

Temporal and Modal Logic

Temporal logic (TL) is a special type of modal logic, it provides a formal way to qualitatively describe and reason the way the truth values of assertions changes over time. Temporal logic is a good formalism to specify and verify the correctness of continuous concurrent computer programs (e.g. OS or network protocols). Temporal logic is a property-oriented method for specifying properties of concurrent and distributed systems, for a given temporal logic inference system, special modal operators concisely state assertions about the system's behavior. Specifiers use these operators to refer to past, current, and future states (or events).

Classification of TL

Propositional vs. First order

Propositional is based on the classic propositional logic, the use of the logic connectives (and, or and not). While First order (FO) Logic is a refined version of propositional logic that includes the use of predicates, variables, constants and quantifiers. A first order logic could be uninterpreted (with no specific stricture definition/assumption) or interpreted (with a specific structure assumption like stacks or integer). A partially interpreted FO TL may have a specific domain but may not define some structure e.g. function symbols.

Global vs. compositional

In endogenous TLs all temporal operators are interpreted in a single universe corresponding to a single concurrent system. Endogenous TLS are suitable for the global reasoning and description of a complete concurrent system.

Exogenous TLs allows temporal operator handles the expression of correctness of program fragments or many small programs in the same system, this allows for compositional or modular program reasoning.

Branching vs., Linear Time

This is simply based on the notion of time in the system interpretation. Time could be viewed as a linear occurrence, in which at a point in time, there exists only one future event. Secondly, time could be viewed as branching, i.e. as a tree like structure, in which time could be split into alternative structures in which there could be different future events.

Points vs. Intervals

The underlying structure of time in a T could be true or false of a point in time or based on intervals of time, the use of times simplifies the formulation of certain correctness properties.

Discrete vs. Continuous

Time could be defined as a discrete time in which the current time represents one state and the next time would be the next state in the system. Continuous timing reveals the state of the system for every possible time in its lifetime; it is applicable for real time systems.

Past vs. Future

Most TL uses are based on describing the occurrence of future events, however there is a need for past event description and evaluation in heuristics or history based systems.

LTS (labeled transition systems)

A labeled transition system consists of states and labeled transitions between states. The states model the system states; the labeled transitions model occurrences of interactions of the system with its environment, e.g., input or output.

Formally, an LTS is a tuple $(S, A, \rightarrow, s_0)$ where:

- S is a (finite) set of states;
- A is a set of actions;
- $\rightarrow \subseteq S \times A \times S$ is a transition relation;
- $s_0 \in S$ is the initial state

CTL Syntax

Let AP be a set of atomic propositions

Inductive construction:

– $\mathrm{A} \in \mathrm{AP}$ is a CTL formula

– Then from CTL formulas ($\phi 1$ and $\phi 2$), we can build

$\neg \phi 1$ $\phi 1 \wedge \phi 2$ EX$\phi 1$ EG$\phi 1$ $\phi 1$ EU $\phi 2$

Where X stands for « next »,

G for « globally », and

U for « until »

CSP

CSP uses a model-oriented method for specifying concurrent processes and a property-oriented method for stating and proving properties about the model. CSP is based on model of truces**,** or event sequences, and assumes that processes communicate by sending messages across channels.

3.4 Tools for formal analysis

JTORX

JTORX was developed and introduced by the University of Twente's Formal methods (FM) group in the Netherlands in 2000. JTORX is a tool to check whether there exists an ioco testing relation between the specification of a system and its implementation. It requires an implementation and its specification as inputs. The specification describes the system behavior (or functionalities) the implementation is allowed to perform. TORX checks the behavior of the implementation in relation to the specification and states its correctness judgment (fail or pass conformance).

LTSA (Labeled transition State Analyzer)

LTSA is a verification tool for concurrent systems. LTSA is formal tool developed by Jeff Kramer and Jeff Magee at the Imperial College London. It checks that the specification of a concurrent system satisfies the properties required of its behavior. LTSA mechanically checks for violations of the properties specified against the specification of the system. The systems and its required properties in LTSA are modeled as finite state machines. This tool supports a process algebra notation, FSP for concise description of the component behavior. The LTSA tool is designed to help modelers formally analyze LTS models (labeled transition systems) written in FSP (finite state processes) to check for safety properties, deadlocks and progress.

NuSMV

NuSMV is a symbolic model checker tool that was designed to improve on the SMV, the original BDD based model checker developed at CMU (Carnegie Mellon University). It is a state of the art model checker tool that can be applicable in verifying systems formally. NuSMV is open source, flexible and well documented and robust to support academic use and possible industrial transfer. It is an extension of SMV and it includes a SAT based model checker to the SMV version. NuSMV implements BDD based symbolic model checking and also integrates model checking techniques based on propositional satisfiability.

PAT

PAT [9] is a self-contained framework for to support composing, simulating and reasoning of concurrent, real-time systems and other possible domains; it comes with user friendly interfaces, featured model editor and animated simulator [9]. Process Analysis Toolkit (PAT) is a symbolic model checker tool developed at the National University Singapore (NUS) in 2007. It is designed to check for properties in systems through specifications written in Communicating Sequential Processes (CSP) and assertions written in LTL and CTL to check properties such as divergence, deadlock, reachability and eventuality. Furthermore, PAT can be used to check for boundedness and satisfiability. Most importantly, PAT can be used to analyze interactions among components of a system, by reviewing properties through the channels that couple the components processes.

Chapter 4: Combining Formal Methods and Simulation

4.1. Importance of combining formal methods with simulation

Formal methods provide a means to mathematically document and analyze systems, through a precise mode of writing requirements and a strong analytical method to assess system properties, structure and behavior. Modeling and simulation on the other hand instead provides a means to infer important information form the structure behavior and properties of systems by creating a model representation of the system and implementing the behavior of such systems from the model.

A combination of these two paradigms provides an avenue to study systems both through a strong analytical/mathematical approach (formal methods) and at the same time provides a realistic approach of viewing the behavior of such system. Formally analyzing simulation helps to provide a robust mathematical background for the simulation process. It is important for the simulation algorithm or program to be formally proven to be correct and versatile through formal methods. Thus, it is important for simulation algorithms to possess a strong formal foundation.

This is pertinent because this combination would provide a formal approach to analyze the models and provide a tool for analytical evaluating performance. This combination makes it possible to formally specify models, automate model analysis, use system requirements and proves system properties, besides formal methods provide the tools necessary for symbolic model reasoning [3].

4.2. Survey of approaches to combining formal methods with simulation

Many scholars in the Modeling and Simulation field have proposed different approaches to combine formal methods and Simulation (especially DEVS). Some scholars advocate for a fully analytical integration of formal methods to simulation, others propose a light introduction of formal methods into simulation formalisms and some scholars propose the formalization of simulation formalisms.

Stevenson et al proposed a direct formalization of DEVS in a mathematical theory, set or category theory [4]. This approach is intended to provide an in depth formal background for DEVS. The advantage of this approach is that it provides an underlying abstract mathematical theory for DEVS and imposes fewer restrictions than a formal method that was not designed for DEVS.

Kuhn et al on the other hand proposed two methods of combining DEVS and formal Methods [5]. The first method is the light weight formal method, which involves both the automation of formal analysis through the use of automatic formal tools, and the focused application of formal techniques that use efficient mathematical analytical methods. The second method is the hiding complexity approach, which propose a means of introducing logic based models into modeling and simulation.

Traore on the other hand proposed a means of combining DEVS with Z specification; logic based formal method [3]. This approach also proposes a means to integrate logic based formal method into the DEVS formalism and also integrate the Z formal method with an already existing formal tool Z/EVES. This was achieved by directly mapping DEVS to the formal representation used by Z/EVES.

4.3. Challenges and the need to use multiple formal methods

There are numerous challenges and limitations surrounding the combination of formal methods with DEVS. This issue includes the limitation of formal methods. Formal methods require the need to master the mathematical skills for intensive analysis and rigorous proof checking. These skills need to be cultivated over time, because errors could be catastrophic and this poses a difficulty for informal DEVS simulation users.

Besides, formal methods are not suitable for non-functional requirements of systems. A formal method cannot express nonfunctional requirements of a system efficiently. Formal methods can express the behaviors of the system but not requirements pertaining to the judgment of the system operation. A formal method can not express the quality goal or the usability criteria of a system. It can only express specific dynamic behaviors of the system and not the way the system is supposed to be, i.e. criteria pertaining to the static structure of the system.

Lastly, no single formal method can fully encapsulate the structural and property variables of DEVS. For example, a state transition based formal method can not capture the logic of a temporal system and a temporal logic based system can not capture the interactions of a process algebra based system. Therefore, there is a need to use many formal methods in this combination in order to capture all aspect of DEVS.

Chapter 5: Formal Framework of DDML

DDML is a simple but expressive graphical notation for DEVS. It possesses a strong formal framework that serves as a strong mathematical foundation for the formalism: its semantics, syntax and property definitions. DDML focuses on three hierarchical levels of abstraction according to the Ziegler's levels of knowledge abstraction, these three levels are namely [6]:

1. Input Output System (IOS) concerned with system dynamic behaviors, especially the change from one state to another and the activities or input stimuli causing such changes as well as the output of the system after each input stimuli. This level is characterized by states and state transitions.

2. Input Output Relation Observation (IORO) is concerned with traces and trajectories of the system i.e. the footprints of the system dynamic behaviors.

3. Coupled Network (CN) is concerned with the structural properties and functional couplings of the system components.

Formally, the DDML semantics are mapped into these three levels of abstraction in order to capture the relevant information important at each level. In order to assess these semantics at each level, different formal methods adaptive to the specialization of each level are employed to help formally express the DDML framework. In view of this, the different tools that correspond to each level of abstraction and semantic definition would also be employed at each level.

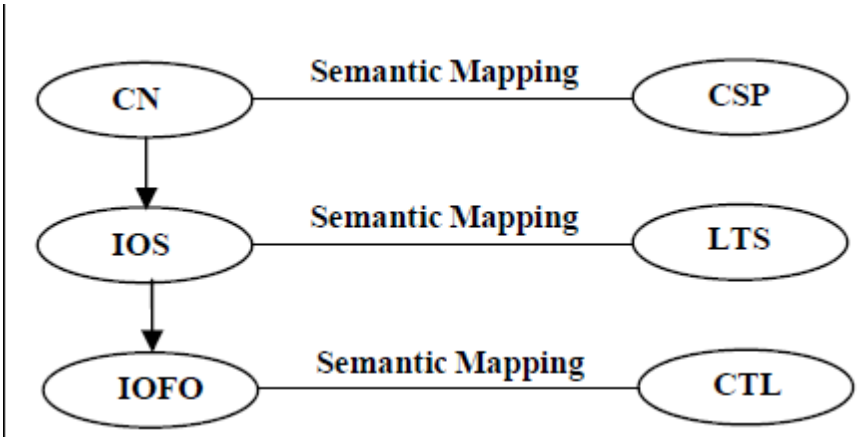


Fig 3: Semantic mapping of DDML hierarchical levels to corresponding formal methods

At the IOS level, the DDML semantics can be mapped to the labeled transition system (LTS) formal method in order to capture the transition based formal definition of the model. At the IORO level, the DDML semantics can be mapped to the Linear Temporal Logic (LTL) and the Computation Tree Logic (CTL) semantics in order to capture the temporal logic based formal definition of the modeling language. Finally, the CN level of DDML can be mapped to the Communicating Sequential Process (CSP) formal method in order to capture the process algebraic form of the coupled model and the interaction of its component parts.

DDML CN level to CSP

The translation of DDML coupled Network (CN) level to Communicating Sequential Processes (CSP) was done by Traore et al through the formalization of the DDML CN [8]. This was achieved by using the Object Z formal specification language to capture the abstract syntax of DDML CN level in order to proffer a simpler and expressive power in logic, then translating the DDML abstract syntax in Objective Z to CSP accordingly.

DDML CN to Objective Z abstract Syntax

The DDML CN level is majorly composed of the Models, Ports, Couplings and the Select function. The translation of these component parts to the simpler Objective Z abstract is defined accordingly below. For each component of the DDML CN a type definition for their names is required, this is shown below [8]:

$$[ModelName, PortName, CouplingName]$$

A port has a name and a direction (in or out corresponding to input or output port types respectively).

$$Direction ::= OUT \mid IN$$

Furthermore, a port is always connected to a model (AtomicModel or CoupledModel). Thus a port is associated with the name of a model. The schema of a Port is

Port

$name$: $PortName$
$mname$: $ModelName$
$direction$: $Direction$

The *Port* schema consists of attributes *name*, *mname*, and *type*. Port names are unique. This implies that there exists a one-to-one mapping between port names and ports. This function $\mathcal{F}$ is defined as

$$\mathcal{F} : PortName \rightarrowtail Port$$

$$\forall n : \mathrm{dom}\, \mathcal{F} \bullet n = \mathcal{F}(n).name$$

The attributes of *Coupling* are *name* (or descriptor) and a non-empty finite set of *ports* (exactly two ports). A Coupling is represented by the schema

Coupling

$name$: $CouplingName$
$ports$: $\mathbb{F}_1 Port$

$\sharp ports = 2$

Also, the names of couplings are unique; hence there exists an injective function *C:*

$$C : CouplingName \rightarrowtail Coupling$$

$$\forall\, n : dom\ C \bullet n = C(n).name$$

A *Model* has a name and a finite set of ports to communicate with its environment. It is represented by the schema

Model

$$name : ModelName$$
$$ports : \mathbb{F}\ Port$$

$$\forall\, p : ports \bullet p.cname = name$$

The predicate implies that all the ports of a model refer to that model.

An *(AtomicModel)* inherits *Model* as such has the same definition as given above.

In addition to name and ports, a coupled model has a finite set of Models (these models can either be AtomicModels or CoupledModels leading to a hierarchical structure) referenced by names, and a finite set of couplings.

CoupledModel **inherits** *Model*

$$models : \mathbb{F}\ Model$$
$$couplings : \mathbb{F}\ Coupling$$

$$\forall\, c : couplings \bullet c.ports \subseteq ports \cup \bigcup \{m : models.ports\} \quad [i]$$
$$\forall\, c1, c2 : coplings \bullet c1.name = c2.name \Leftrightarrow c1 = c2 \quad [ii]$$
$$\forall\, c1, c2 : couplings \bullet c1.ports = c2.ports \Leftrightarrow c1 = c2 \quad [iii]$$
$$\forall\, c : couplings \bullet \forall\, p1, p2 : c.ports \bullet p1 \neq p2 \quad [iv]$$
$$EIC \subseteq couplings \wedge EOC \subseteq couplings \wedge IC \subseteq couplings \quad [v]$$
$$EIC \cap EOC = EIC \cap IC = EOC \cap IC = \emptyset \wedge \bigcup \{EIC, EOC, IC\} = couplings \quad [vi]$$
$$\forall\, c : couplings \bullet \forall\, p1, p2 : c.ports \bullet (p1 \neq p2 \Rightarrow (p1 \in ports \wedge p2 \notin ports \Leftrightarrow p1.direction = p2.direction)) \quad [vii]$$
$$\forall\, c : couplings \bullet \forall\, p1, p2 : c.ports \bullet (p1.direction = p2.direction \wedge p1 = Direction.IN) \Leftrightarrow c \in EIC \quad [viii]$$
$$\forall\, c : couplings \bullet \forall\, p1, p2 : c.ports \bullet (p1.direction = p2.direction \wedge p1 = Direction.OUT) \Leftrightarrow c \in EIC \quad [ix]$$
$$\forall\, c : couplings \bullet \forall\, p1, p2 : c.ports \bullet (p1 \neq p2 \Rightarrow (p1, p2 \notin ports \Leftrightarrow p1.direction \neq p2.direction)) \Leftrightarrow c \in IC \quad [x]$$

The select function is a function that selects a given model out of a finite set of imminent models. The select function is a sequence of models, with a decreasing order of priority.

Select

models: $\text{seq}_1 Model$

Abstract Syntax to CSP

Furthermore, Traore et al translate the abstract syntax of DDML to CSP with the following definitions [8]:

An arbitrary port of a DDML model defines a logical point of interaction between the model and its environment. A port typically represents the set of interaction events with the environment through its interface. Hence, ports in DDML can be mapped onto CSP compound events. These events are realized through CSP channels. We define a translation function ψ that maps a DDML port to CSP channel:

$$\psi : Port \rightarrow Channel$$

Hence, if c is the name of a port and T is the type of object communicated down it, we would have that

$$\psi(p) = c.T = \{c.x \mid x \in T\} \subseteq \Sigma$$

A model in DDML is defined by a set of ports (input and output ports) and a component specification that represent the abstract behavior of the model. Since models can have many ports, ports allows a model to define multiple interface to its environment. A DDML model can be mapped to a CSP process. Recall that at the coupled network abstraction level in DDML, the precise internal behavior of model is not shown. Hence, we map a DDML model to an undefined CSP process with hidden internal events. The alphabet contains only external events (or channels). We define a function $\mathcal{M}$ that maps a DDML model to CSP process

$$\mathcal{M} : Model \rightarrow Process$$

Hence, if A_m is a DDML model and $S_{internal}$ represents the set of all internal events,

$$\mathcal{M}(A_m) = P_A \setminus S_{internal} = A_p$$

Where P_A is an undefined process and the alphabet of A_p is given as

$$\alpha A_p = \bigcup\{\psi(p) \mid p \in ModelPorts\}$$

A coupled model in DDML is composed of several interacting models. These models are coupled or connected via port attachments or couplings. Communications between the models take place between ports that are compatible. Such communication is synchronous and can be generalized as follows.

Coupled models belong to the domain of the function $\mathcal{M}$. Hence, If *C* is a coupled model there exist an equivalent undefined CSP process.

$$\mathcal{M}(C) = C_p$$

Let *Sub* be the set of all sub-models of the coupled model *C.* Then

$$V = \|_{P:Sub} \ (P[\![R_C]\!], X_p)$$

V is the parallel composition of all the sub-models. This is the *n*-way parallel composition of all the sub-models in *Sub*.

$$C_p = V\,[\![R_{EC}]\!]) \setminus H_{IC}$$

Where

$P[\![R_C]\!]$ is a renaming operation that changes port names to coupling names:

X_p represents the alphabet of *P.* This is the set of external channels (after renaming).

$V\,[\![R_{EC}]\!]$ is the renaming operation used to rename coupling names from a sub-model to the an external port of the coupled model.

H_{IC} denotes all the internal couplings that are hidden.

DDML IOS Level

The DDML IOS Model is mapped to the Labeled transition system (LTS) formal Method. The LTS as earlier defined in chapter 3 is a tuple of the form (*S*, Λ, →), where *S* is a set (of states), Λ is a set (of labels) and → ⊆ *S* × Λ × *S* is a ternary relation (of labeled transitions) [8].

Meanwhile, an atomic DDML model is a tuple of the form [8]:

B = <XB, YB, SB, S0B, ψ, C (VB), Tint, Text, C(e), OP, ϕ, Act >

Where

- X_B = p, X_p |p ∈ IPorts and dom p = X_p. Such that #X_B < +∞ is the set of input ports;
- Y_B = q, Y_q |q ∈ OPorts and dom q = Y_q . Such that #Y_B < +∞ is the set of output ports;
- V_B = v, S_v |v ∈ StateVar and dom v = S_v #V_B < +∞ is the set of state variables;
- S_B is a finite set of state configurations;
- $S_{0B} \in S_B$ is the initial state of B;
- OP is the set of operations defined on the state variables;
  Given that "ops" is an operation and "a" is a state, the notation a.ops indicates that "a" is operated by "ops" to change the state of "a".
- (V_B) is a set of constraints on state variables. A constraint is of the form

- c ≔ v ∈ D|v ∈ D ∧ v′ ∈ D′|v ∈ D ∨ v′ ∈ D′ . Where $v = v_1, \dots, v_n \; and \; D \subseteq dom\, v_1 \times \dots \times dom(v_n)$,
- v′ = v′$_1$, … , v′$_m$ and D′ ⊆ dom v′$_1$ × …× dom(v′$_m$) .
- ψ: $S_B \rightarrow \mathbb{P}(V_B)$ is a mapping between each element of S_B and a finite set of conditions on the variables in V_B. Given s ∈ S_B, We denote $s = s' \in dom\, S_v \; v \in StateVar \mid s' \vDash \psi(s)$
- The function ψ is defined by the following: $\psi \; S0B$ is verified by the initial state of the system. s s∈SB = dom Sv v∈StateVar and ∀s, s′ ∈ SB, s ⋂ s ′ = ∅ for $s \neq s'$
- Act is the set of activities;
- φ: $SB \rightarrow \mathbb{P}\; Act$ is the mapping from states to the set of activities ;
- T_{int} ⊆ (SB × ℝ+) × C(VB) × Y × ℙ OP × (SB × ℝ+) is the set of internal transitions satisfying: The internal transition s, d , c, l, ops, (s′, d′) ∈ Tint will be denoted by s, d <c,l,ops > iB s′, d′ or s, d <c,l,ops > s′, d′ to avoid confusion.
- $C(e)$ is the set of conditions α of the elapsed time e of the for
- $\alpha \coloneqq e \sim t \mid e - t \sim t' \mid e + t \sim t' \mid e * t \sim t' \mid \alpha 1 \wedge \alpha 2 \mid \alpha 1 \vee \alpha 2$ Where~ ∈ =, ≤,<, ≥,> , e, t, t' are positive real numbers and $\alpha 1$ et $\alpha 2$ are conditions (denoted by ⊨ α if e satisfies the condition α);
- T_{ext} ⊆ (SB × ℝ+) × C(VB) × X × C(e) × ℙ OP × (SB × ℝ+) is the set of external transition.
- The internal transition s, d , c, x, c e , ops, (s′, d′) ∈ Text will be noted s, d <x,c(e),ops > eB s′, d′ or s, d <x,c(e),ops > s′, d′ . In particular, if there is no branching condition, the transition s, d , ∅, x, c(e), ops, (s′, d′) ∈ Text will be denoted s, d <x,c e ,ops > eB s′, d′ .

At a given instance of time, the system is in a configuration $s \in SB$ with a life span d that is to say the variables satisfy ψ s . If the lifespan of the current atomic state $s1 \in s$ elapses before an external event occurs then the model output l is sent just before transiting to another cluster state s' such that $s, d, l, ops, (s', d') \in Tint$ (internal transition, $v.\, ops \vDash \psi(s'')$). When an external event x occurs before the end of the lifespan of the state, the model transits to the cluster state s' such that $s, d, x, c\, e, ops, (s', d')$ (external transition, $v.\, ops \vDash \psi(s')$ with a life time of d".

Given an atomic DDML model

B = <XB, YB, SB, S0B, ψ, C(VB), Tint, Text, C(e), OP, φ, Act >

It can be shown that B is equivalent to LTS

$L\,A =< SL, init, \Sigma, D, T >$

Where,

$SL = Q = \{((s, e), d) \mid s \in S_B \; and \; 0 \leq e \leq \}$;

- $init = ((S_{0B}, 0), d)$ is the initial state;
- $\Sigma = (\biguplus_{p \in IPorts} X_p) \cup (\biguplus_{q \in OPorts} Yq) \cup (\mathbb{R}^+ \cup \{0, +\infty\})$ is the set of events (alphabet);
- $D = \{((s, e), d) \xrightarrow{y} ((s', 0), d') \mid \exists ops \in \mathbb{P}OP, e = d \; and \; ((s, d), y, ops, (s', d')) \in Tint\} \cup$
- $((s, e), d) \xrightarrow{x} ((s', 0), d') \mid \exists ops \in \mathbb{P}OP, 0 \leq e < d \; and \; ((s, d), x, c(e), ops, (s', d')) \in Text\}$
- $T = \{((s, e), d) \xrightarrow{t} ((s, e'), d) \mid e' = e + t < d\}$

In addition, informally we present a tabular translation of the DDML IOS model to LTS.

DDML IOS	LTS	Interpretation
S_B	S	Finite set of states
Act	A	Set of activities
S_{OB}	S_o	Initial state
T Defined by : $S \times \boldsymbol{A} \times S$ Where: $S \in S_B$ $A \in \boldsymbol{Act}$	$\rightarrow$ Defined by : $s \times a \times s$ Where: $s \in S$ $a \in A$	Transition relation

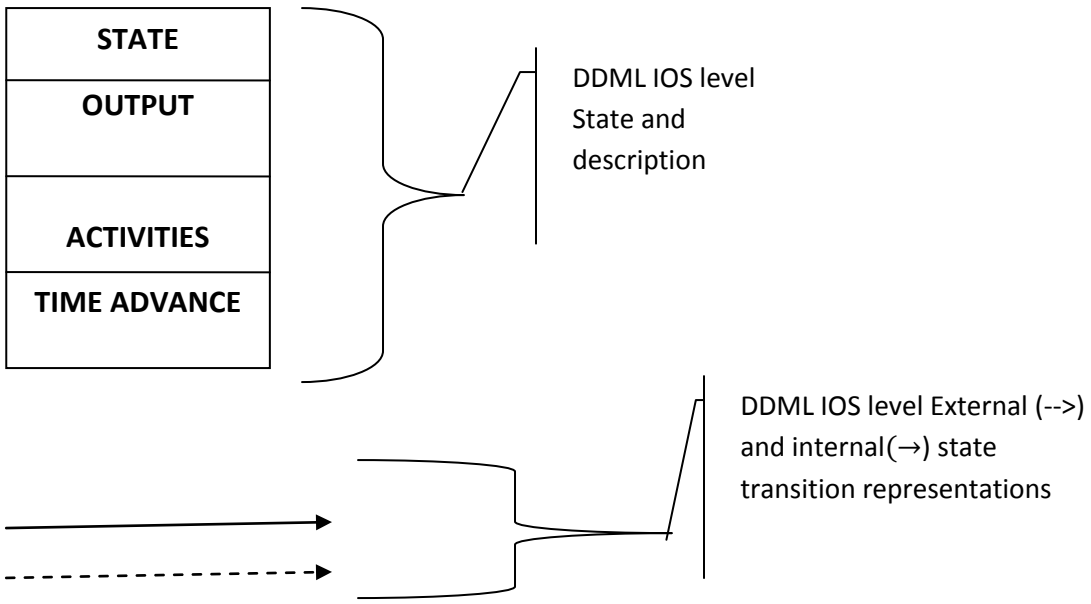
Table 1: DDML IOS to LTS semantic translation

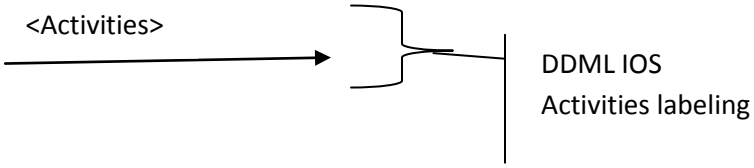
From the above table, it is important to note that LTS is limited and cannot capture all the aspects of DDML IOS elements (like the timing aspects elapsed time, time advance, output events, conditional transition.

Graphical Mapping

Furthermore, a graphical mapping exercise is carried out to show the diagrammatic relationship between DDML and LTS. Considering the fact that both modeling languages have graphical representations, it is important to assess these graphical semantics, their equivalence and their relevance at the DDML system level.

The DDML IOS level semantics involve the use of the following components:





On the other hand, the LTS graphical representation is shown below:

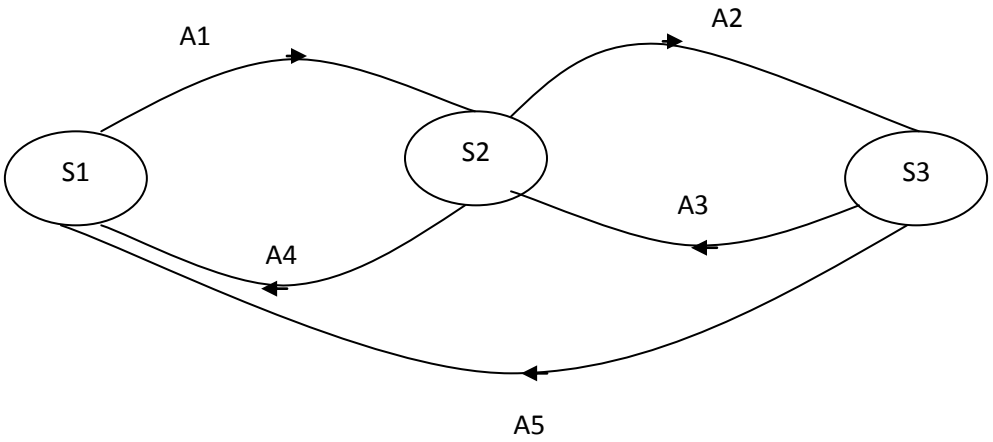


Fig. 4: Example LTS diagram

Key:

S = (S1, S2, S3): represent the set of states

A = (A1, A2, A3, A4, A5): represent set of actions/activities

Mapping the DDML IOS diagrammatic component representation to the LTS graphical representation, the following representation is obtained.

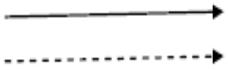
DDML IOS graphical representation	LTS diagrammatic representation equivalent	Interpretation
STATE OUTPUT ACTIVITIES TIME ADVANCE	State name	Graphical mapping of state representation from DDML IOS to LTS
		State transition in DDML IOS to LTS
<Activities>	Activity name 	Activity labeling in DMML IOS to LTS

Fig. 5: Graphical Mapping DDML IOS to LTS

DDML IORO Level

Formally a DDML IORO is a tuple of the form <T, X, Ω, Y, R >, where:

- <T, X, Y> is the observation frame (T-Time, X-Input, Y-Output)
- Ω is the set of all possible input segments
  - $\Omega \subseteq (X, T)$
- R is the I/O relation
  - $R \subseteq \Omega X (Y, T)$

- $(\omega, \rho) \in R \Rightarrow dom(\omega) = dom(\rho)$
  - ω: $<t_i, t_f> \rightarrow X$: input segment
  - ρ: $<t_i, t_f> \rightarrow Y$: output segment

However, in the course of this work, the DDML Atomic Model would be the reference point in the translation and interpretation of DDML IORO to the formal Methods: LTL and CTL. The DDML Atomic Model would be used as the specification language to capture the system properties at this level, while LTL and CTL would define the property checking semantics. This is because CTL and LTL semantics are limited and can not capture all the aspects of the DDML Input Output Relation Observation framework, but can capture the DDML Atomic model through the Kripke Structure definition.

Furthermore, the importance of the DDML atomic model is due to the fact that it is a composite structure of the DDML IORO system. The DDML IORO combines both the Input output relation and the state transition system to describe models in the DDML formalism. In addition, graphically the DDML IORO is highly state transition based. Thus, in order to achieve this work, we focus on the state transition section part of the DDML IORO which in turn represents the DDML Atomic model.

Formally, a DDML Atomic Model is a tuple of the form:

B = <XB, YB, SB, SOB, ψ , C(VB), Tint, Text, C(e), Op, φ, Act >

- XB: set of input ports
- YB: set of output ports
- SB: finite set of state clusters
- SOB: initial state of B
- Ψ: SB $\rightarrow \mathbb{P}\, C\,(\mathrm{VB})$: is a mapping between each element of SB and a finite set of conditions on the variable in VB i.e. a mapping between the states and the state variable conditions
- C (VB): set of Constraints on state variables
- Tint: is the set of internal transitions
- Text: is the set of external transitions
- C (e): is the set of conditions α of the elapsed time e
- Op: is the set of operations defined on the state variables
- φ: SB $\rightarrow \mathbb{P}\, Act$: is the mapping from states to the set of activities
- Act: is the set of activities

Kripke structure is similar to the notion of transition system. In relating the Kripke Structure to the transition system, one can see that every Transition system (in this case state transition system) can be considered as a Kripke Structure by focusing on the different states in the system and the actions that cause transition among states.

A Kripke Structure is a tuple K = <S, S_o,→, L > where:

- S : is a set of States
- $S_o \in S$: is a designated initial state
- → : S x S is a transition relation

- L: S → 2^p is a labeling function

Given an atomic DDML model

B = <XB, YB, SB, S0B, ψ, C(VB), Tint, Text, C(e), OP, φ, Act >

It can be shown that B is equivalent to Kripke Structure

K = <S, S_0,→, Σ, L>

Where:

- $S = Q = \{((s, e), d) \mid s \in S_B \; and \; 0 \leq e \leq \}$;
- $S_0 = ((S_{0B}, 0), d)$ is the initial state;
- $\Sigma = (\biguplus_{p \in IPorts} X_p) \cup (\biguplus_{q \in OPorts} Yq) \cup (\mathbb{R}^+ \cup \{0, +\infty\})$ is the set of events (alphabet);
- $L = \{((s, e)\ , d) \xrightarrow{y} ((s', 0)\ , d') \mid \exists ops \in \mathbb{P}OP, e = d \; and \; ((s, d), y, ops, (s', d')) \in Tint\} \cup$
- $((s, e)\ , d) \xrightarrow{x} (\ (s', 0)\ , d') \mid \exists ops \in \mathbb{P}OP, 0 \leq e < d \; and \; ((s, d)\ , x, c(e)\ , ops, (s', d')) \in Text\}$
- $\rightarrow = \{((s, e)\ , d) \xrightarrow{t} ((\ s, e')\ , d) \mid e' = e + t < d\ \}$

DDML IOS	Kripke Structure	Interpretation
S_B	S	Finite set of states
Act	Σ	Set of activities (events)
S_{OB}	S_0	Initial state
T Defined by : $S \times \boldsymbol{A} \times S$ Where: $S \in S_B$ $A \in \boldsymbol{Act}$	→ Defined by : $s \times a \times s$ Where: $s \in S$ $a \in A$	Transition relation

Table 2: DDML IOS to Kripke structure semantic translation

Chapter 6: Formal Analysis with DDML: A Case Study

DDML (DEVS modeling language) is a graphical and hierarchical formalism that is developed to construct models of dynamic systems for simulation. DDML is motivated by DEVS and it is developed to capture the structure and behavior of systems by focusing on the three levels of abstraction, the Input Output System (IOS) level, the Coupled Network (CN) level and the Input Output Relation Observation (IORO) level.

Tools

The tools used in this work are LTSA (labeled transition systems analyzer) and JTORX. LTSA is formal tool developed by Jeff Kramer and Jeff Magee at the Imperial College London. These tools are designed to help modelers formally analyze LTS models (labeled transition systems) written in FSP (finite state processes) to check for safety properties, deadlocks and progress.

JTORX was developed and introduced by the University of Twente's Formal methods (FM) group in the Netherlands in 2000. JTORX is a tool to check whether there exists an ioco testing relation between the specification of a system and its implementation. It requires an implementation and its specification as inputs. The specification describes the system behavior (or functionalities) the implementation is allowed to perform. TORX checks the behavior of the implementation in relation to the specification and states its correctness judgment (fail or pass conformance).

6.1 DDML IOS Level Analysis

The focus at this level is the IOS level; this level is concerned with the characteristics of a component and its operations. These operations are modeled as state machines and transitions in DDML. The IOS level describes the system through its attributes and operations. The DEVs equivalent abstract relation of the IOS level is the atomic Model. The atomic model is always in a particular state or the other; these states are being represented in the DDML IOS models and their state transitions.

The case study used in the course of performing this task is the traffic light case study. The traffic light case study is expected to model the traffic light system and the IOS model diagram as well as the LTS translation is shown below (fig. 6 & 7).

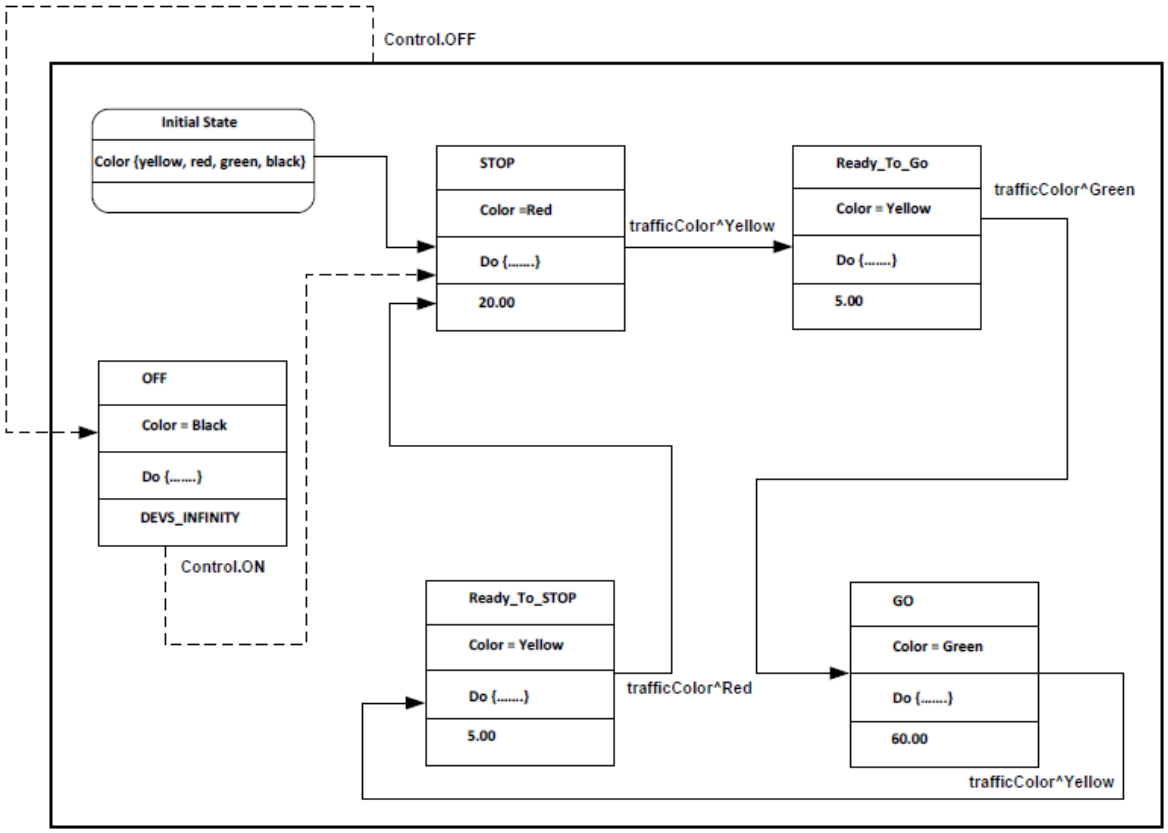


Fig 6: Traffic Light DDML IOS level model diagram

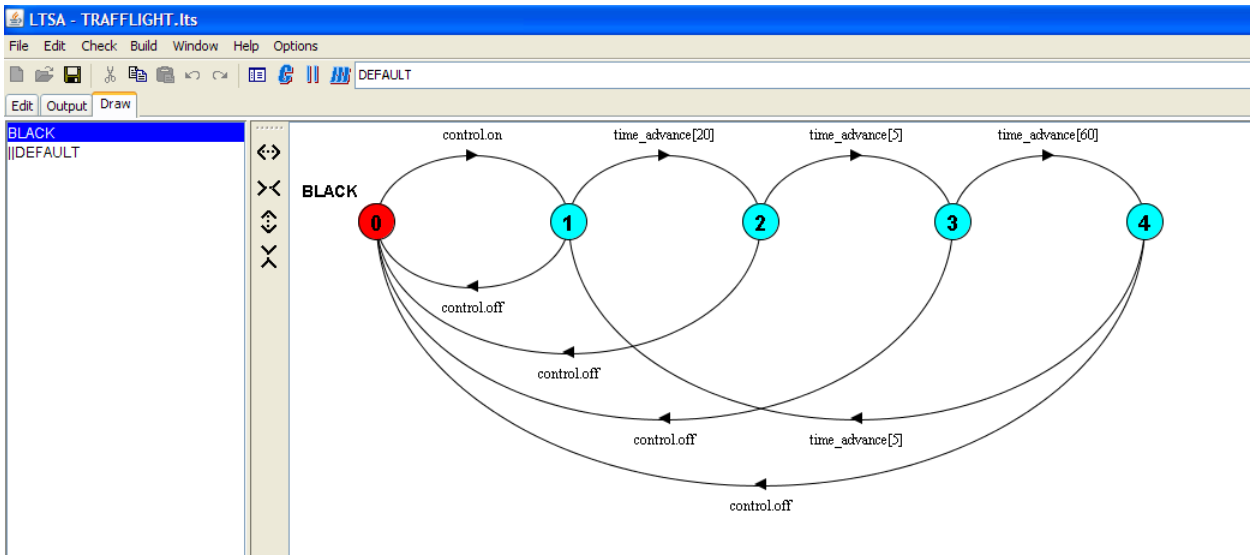


Fig 7: Traffic Light DDML IOS level LTS diagram

LTSA Tests

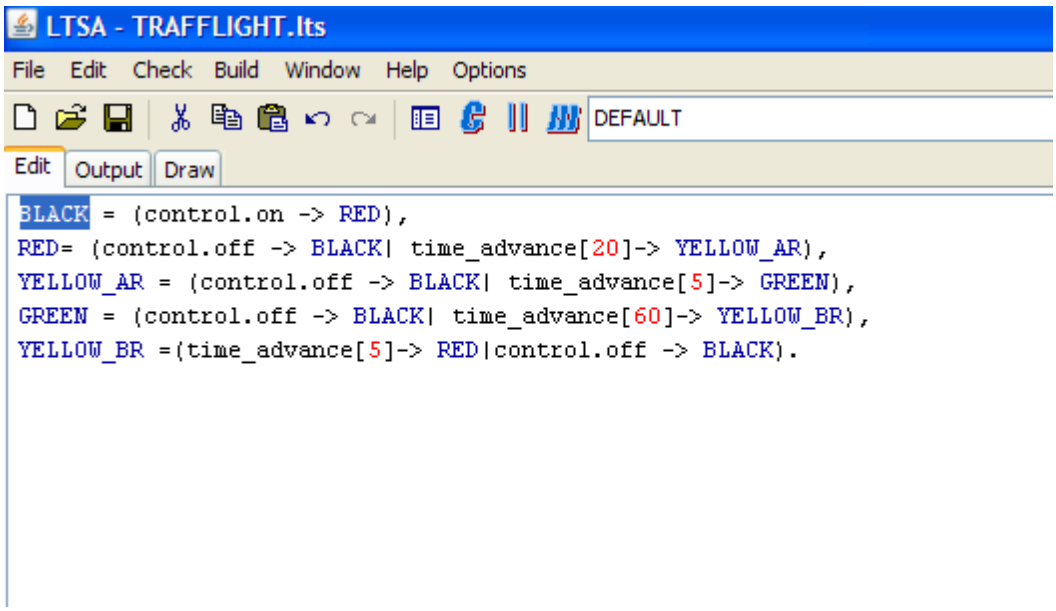


Fig 8: FSP code for Traffic Light DDML IOS

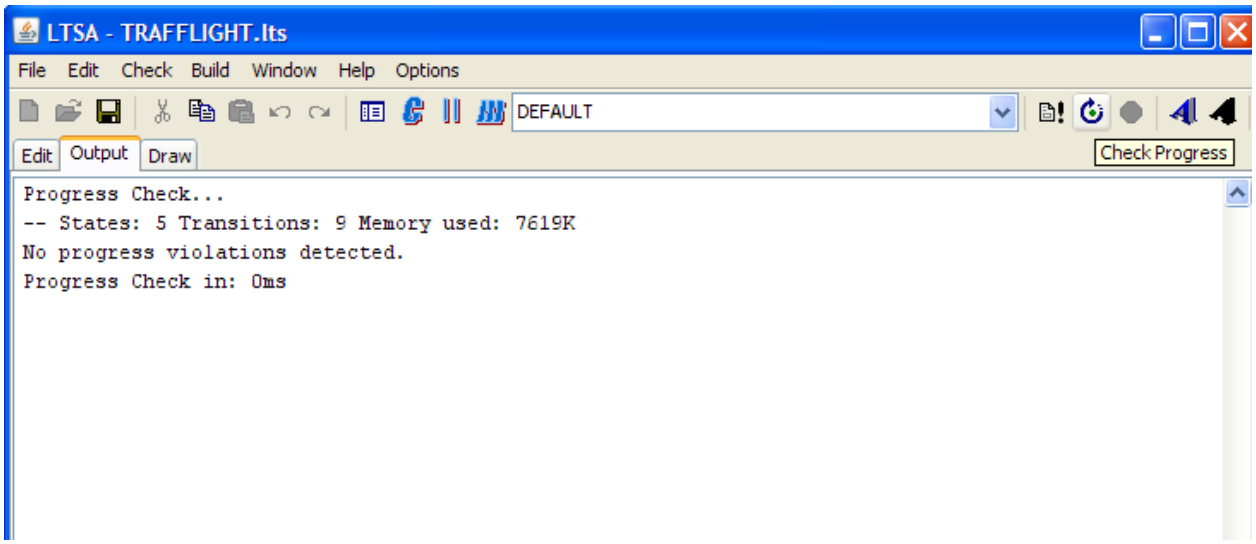


Fig 9: Progress Check of Traffic Light system

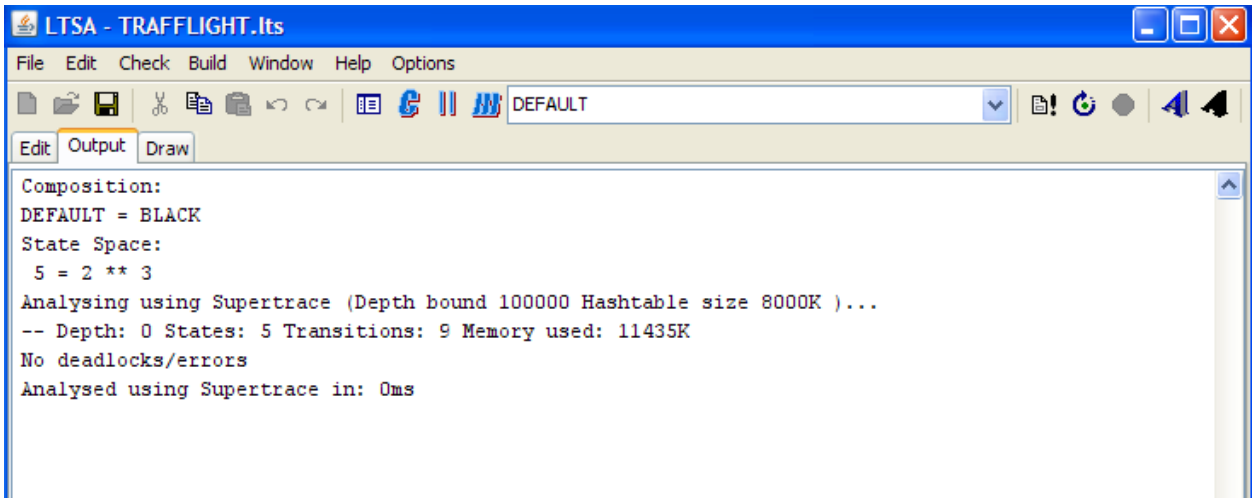


Fig 10: Safety check of traffic Light system

Properties Examined

Progress: Following the DDML IOS model of the traffic light (figure 6), the FSP (figure 8) formally (mathematically) writes the system specification of the traffic light system, using the LTSA formal tool, the LTS diagram is generated as shown in figure 7 and the progress property of the system is checked. From figure 9 one can see that the DDML models of the traffic light passed the progress check. Furthermore, figure 9 reveals that the DDML model of the traffic light system, written in FSP passes the safety check. Therefore, considering this case study, one can tell that the system specification written in DDML IOS passes the progress property.

Safety (deadlocks and errors): Safety properties in formal methods, refers to the ability of the system specification not to lead to a deadlock, or the systems ability to avoid errors, i.e. errors are unreachable given the system specification. In addition, the LTSA tool helps confirm the DDML IOS model safety properties. Following the FSP system specification of the Traffic light system in figure 8 the LTSA tool confirm the lack of deadlocks and errors (conversely the conformance of safety property) in this case study as shown in figures 10. Invariably, considering the results of safety check, one can deduce the DDML IOS model (figure 6) passes safety properties.

Recommendation/Deductions

Following the formal analysis performed at the system level, using the LTSA formal tool, one can conclusively deduce that DDML IOS models can be tested for both the safety and progress properties. This is shown in the formal checks performed above.

However, for the traffic light system, one may ask what happens to the system when it is in states 1, 2, 3 or 4 (RED, YELLOWBR, GREEN or YELLOWAR) and it receives CONTROL ON as input, even though following the specification an expected state should follow such input action. But this is not specified in the DDML IOS model (figure 6) and is also lacking in its FSP translation (figure 8) and its LTS interpretation (figure 7). This input stimulation should be specified in the system and may reveal lack of

completion in the model. This shows lack of completion in the system and should be considered and refined in the DDML model system specification. In conclusion, the completion property of DDML IOS model system specification should be revisited and consequently verified appropriately.

Traffic light JTORX Tests

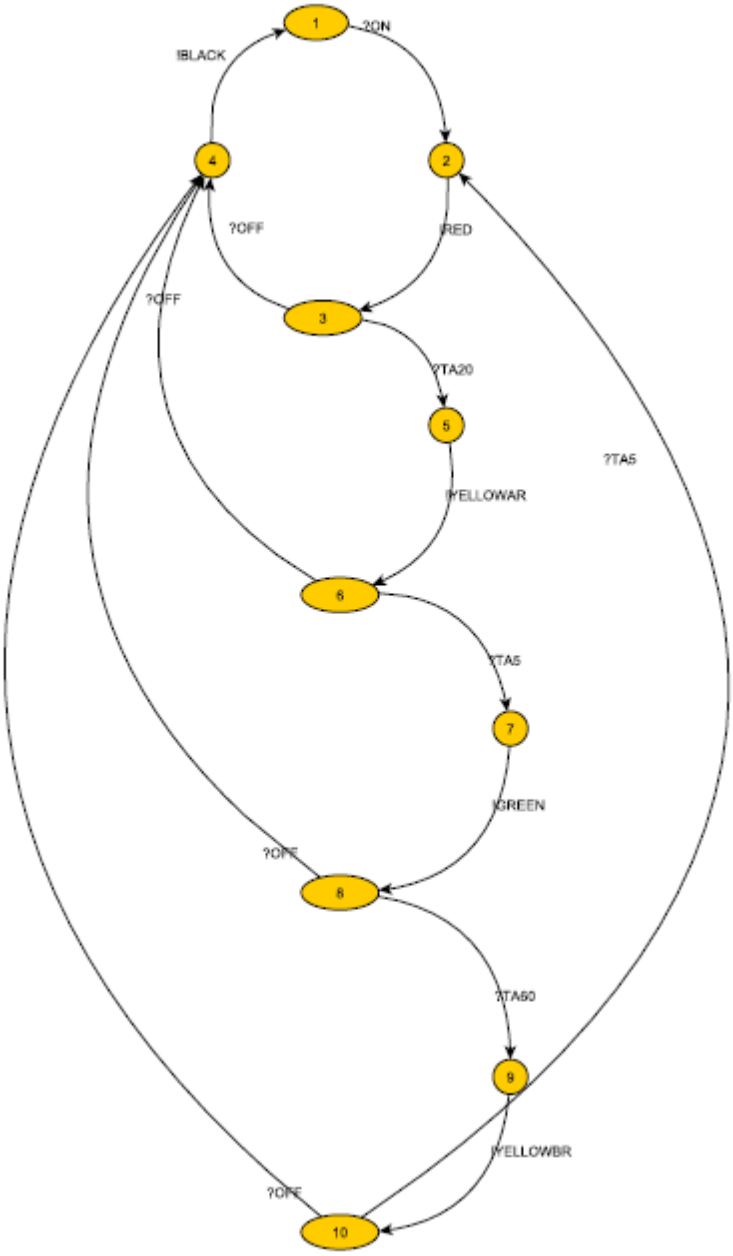


Fig. 11: light specification

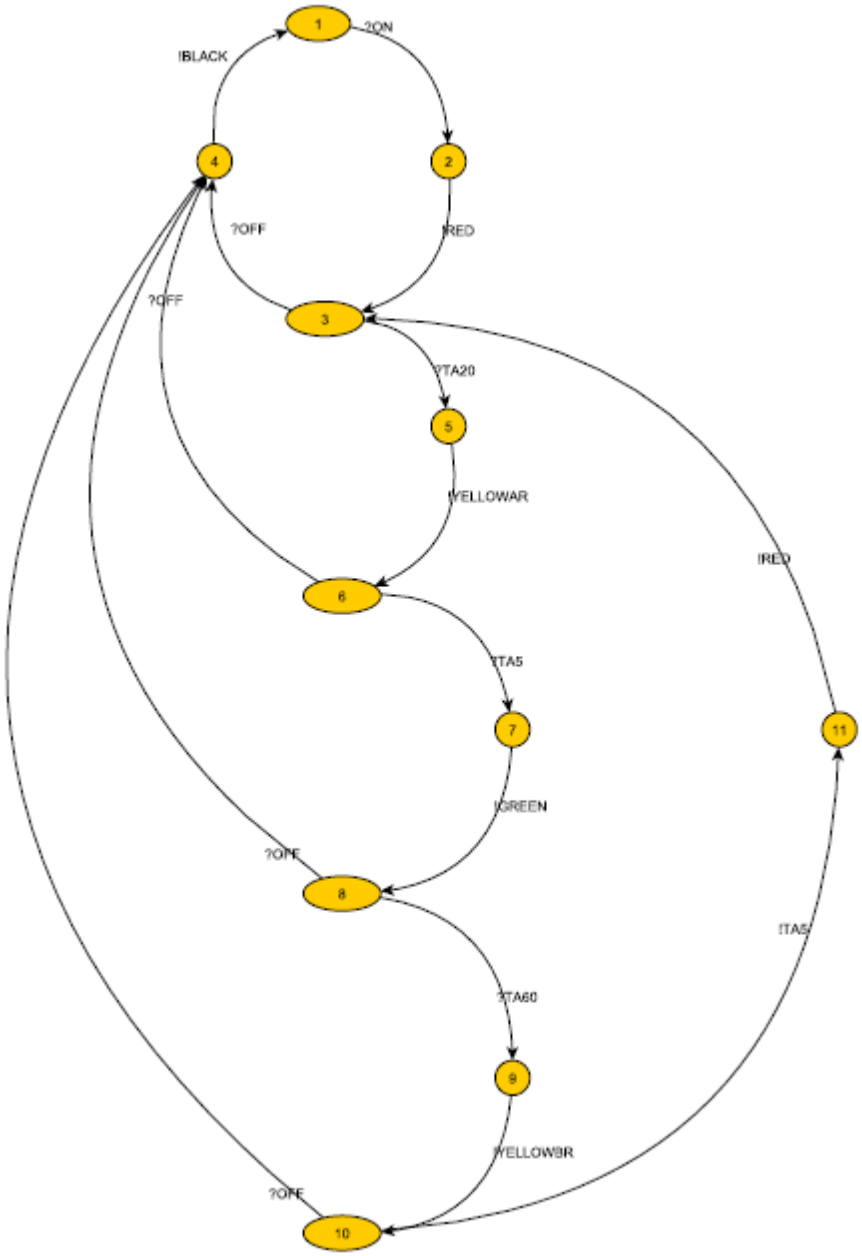


Fig. 12: light implementation 1

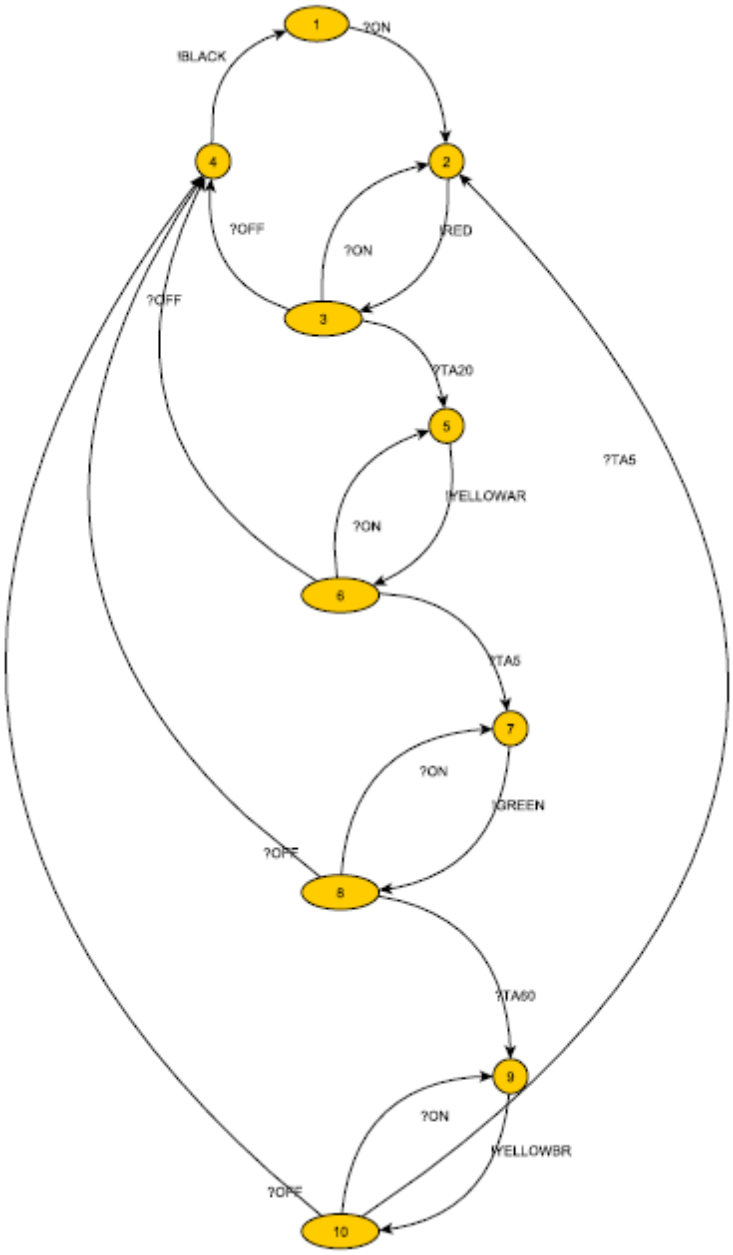


Fig. 13: light implementation 2

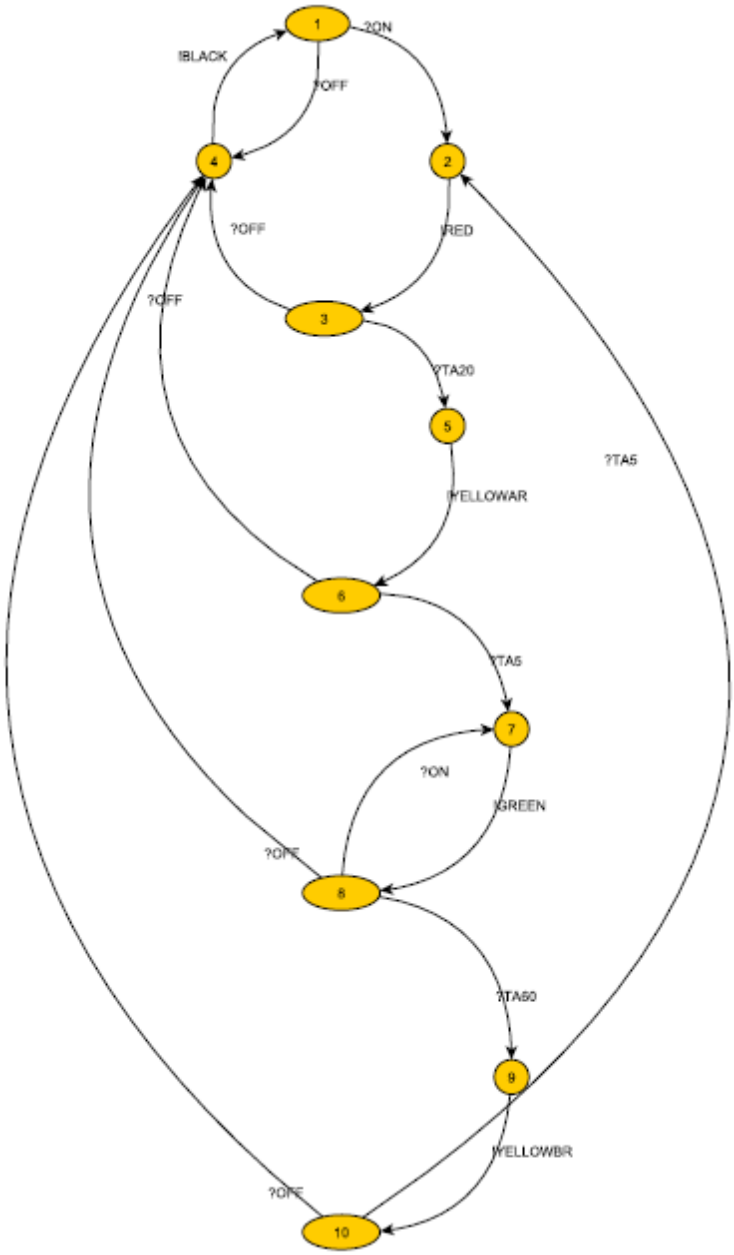


Fig. 14: light implementation 3

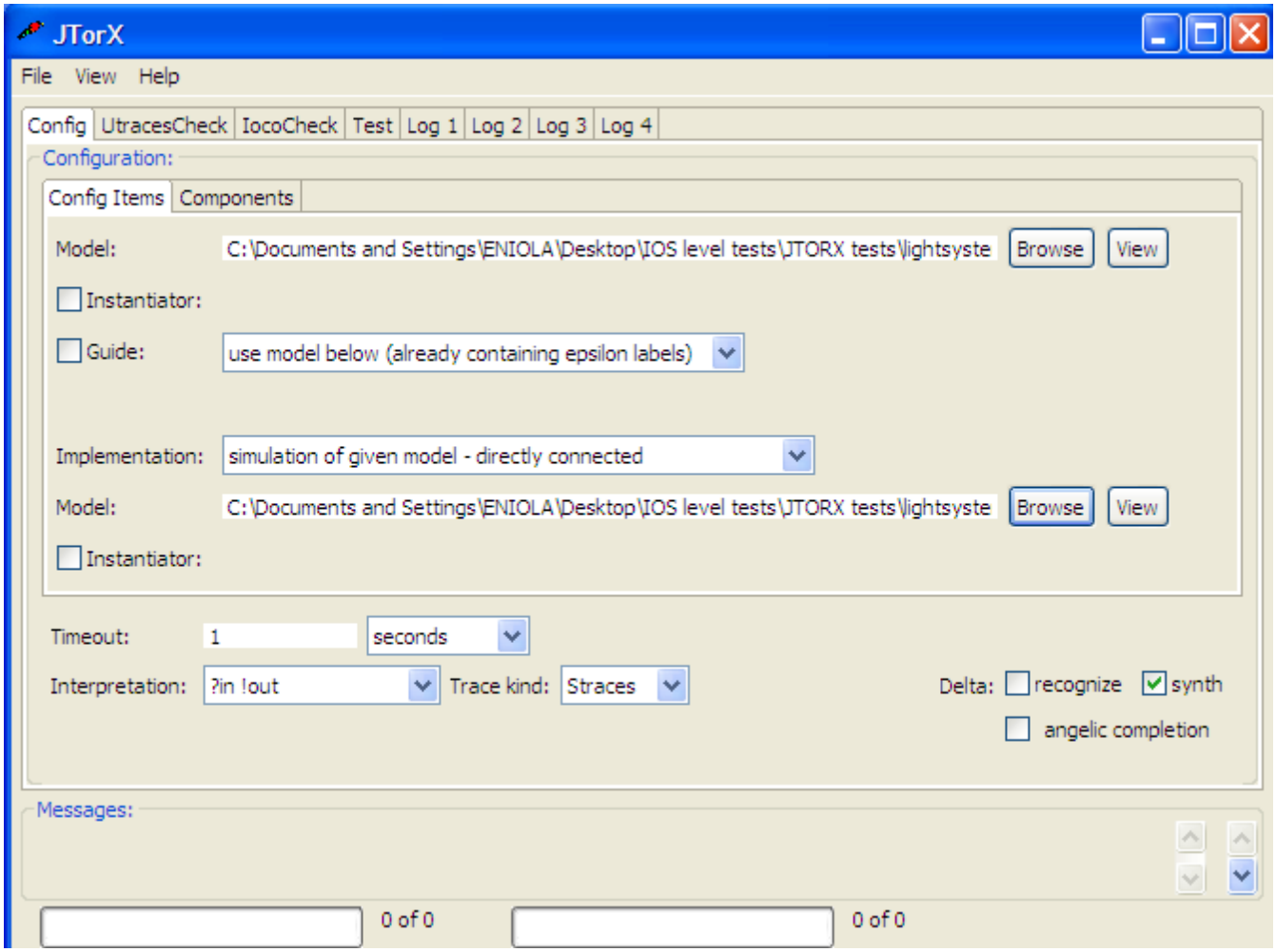


Fig. 15: JTORX configuration

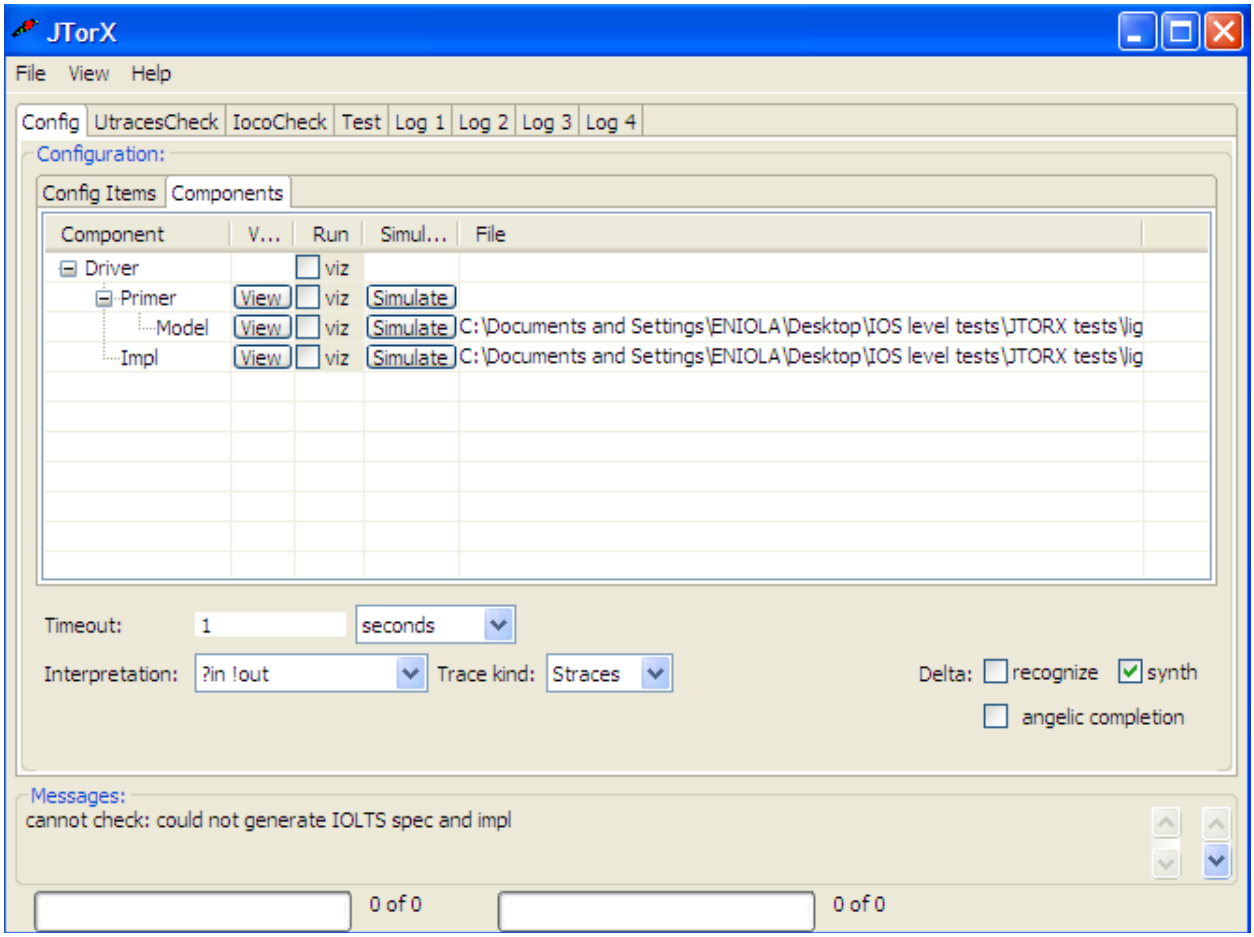


Fig. 16: JTORX component view

Traffic Light tests: Implementation 1

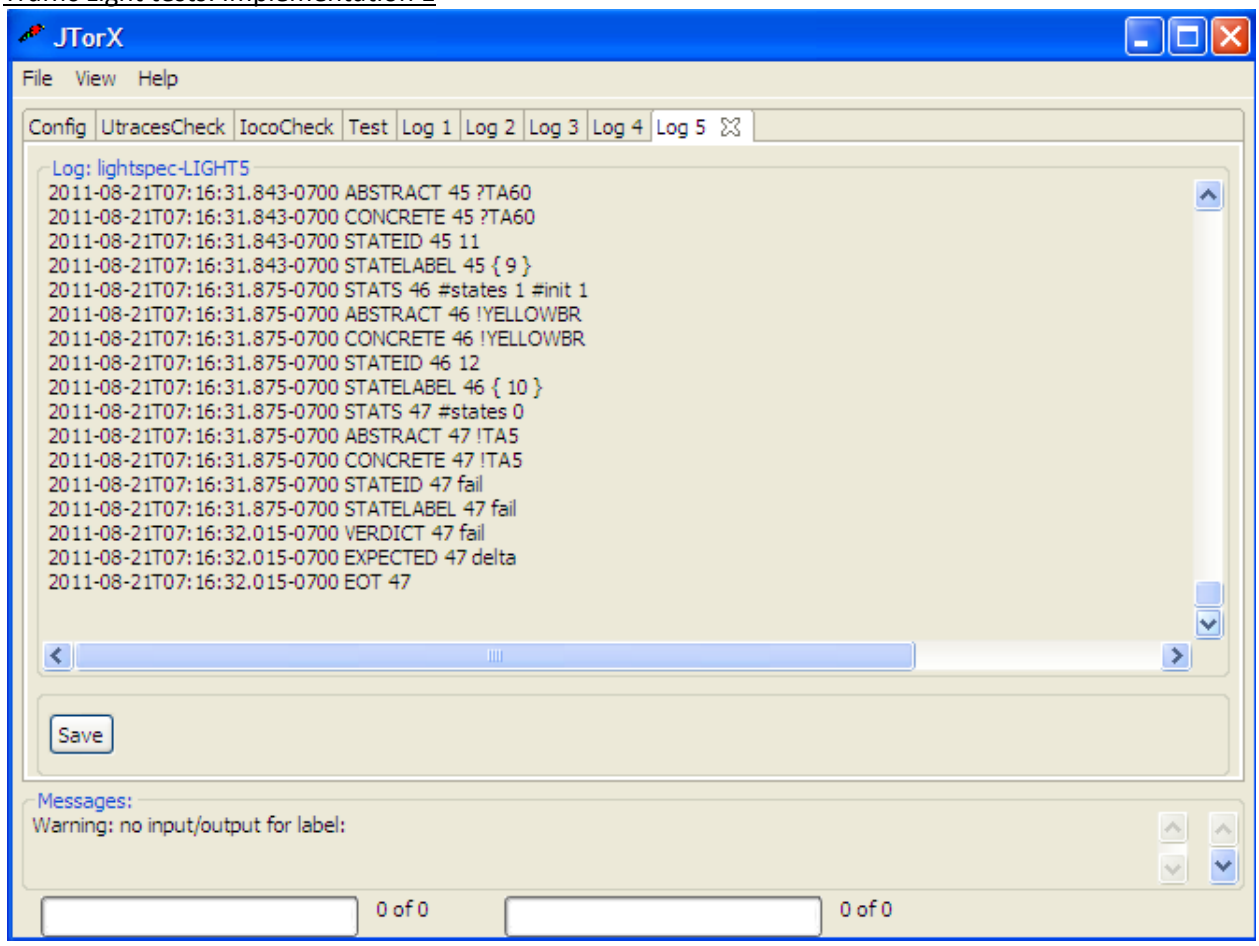


Fig. 17: Implementation 1 log

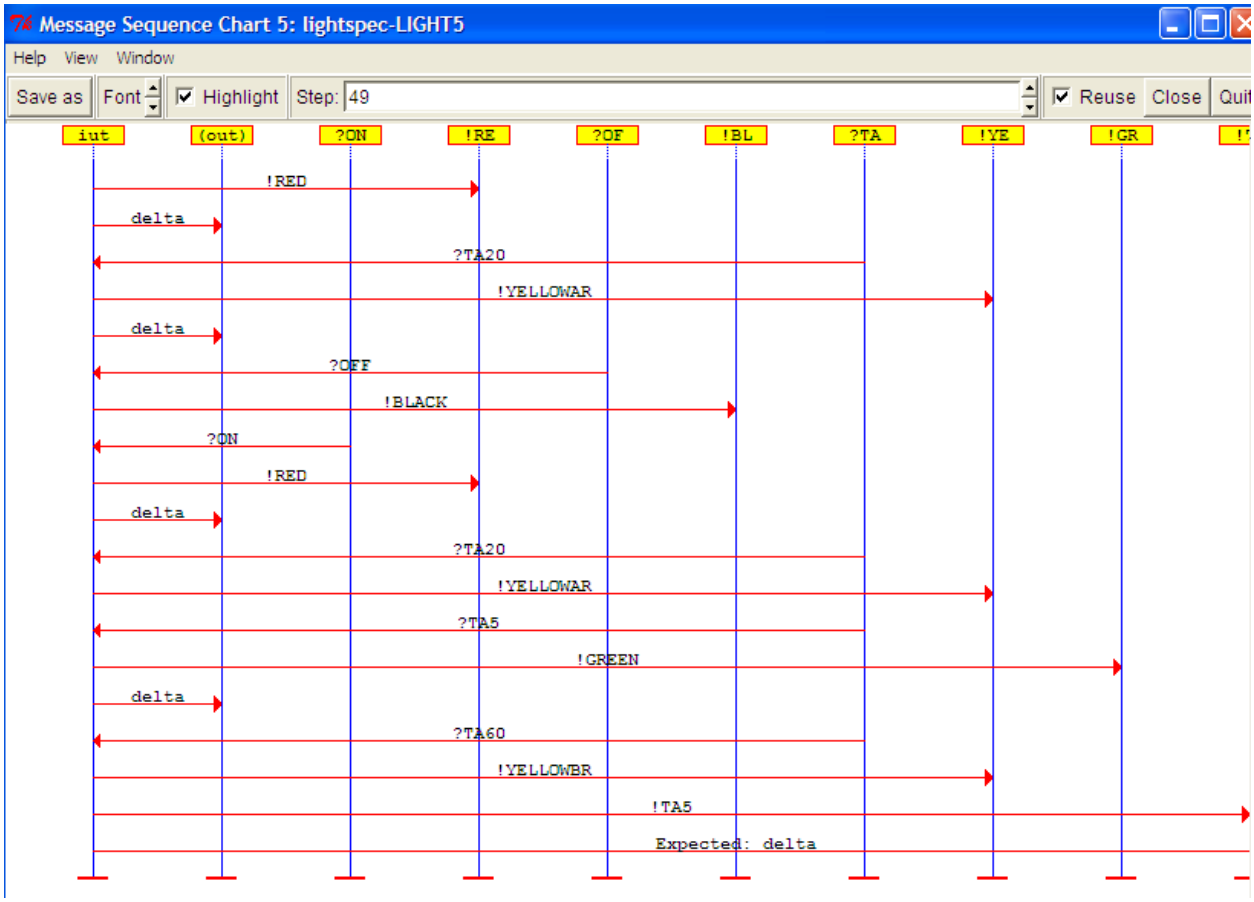


Fig. 18: Implementation 1 message sequence chart

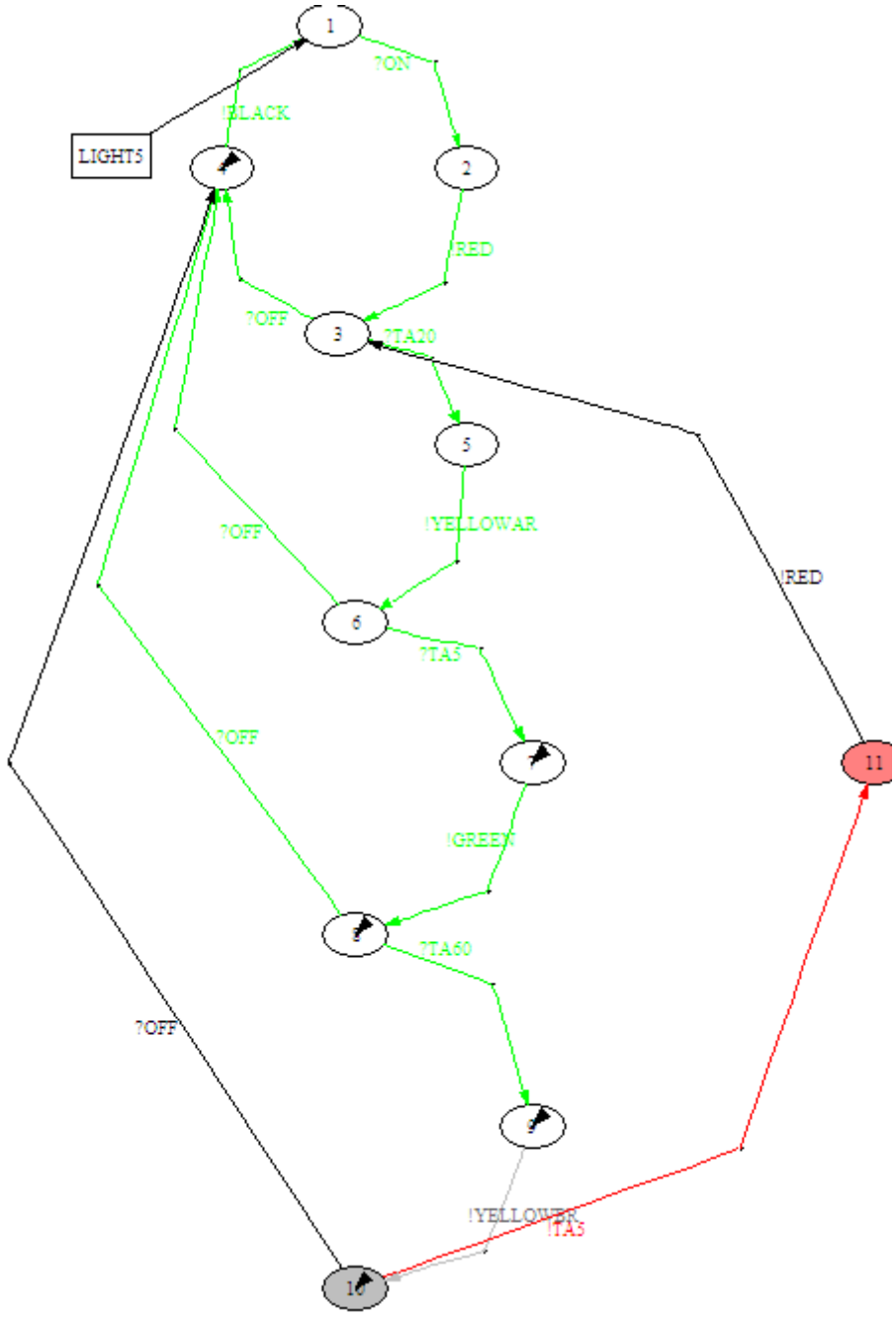


Fig. 19: Implementation 1 antidot implementation

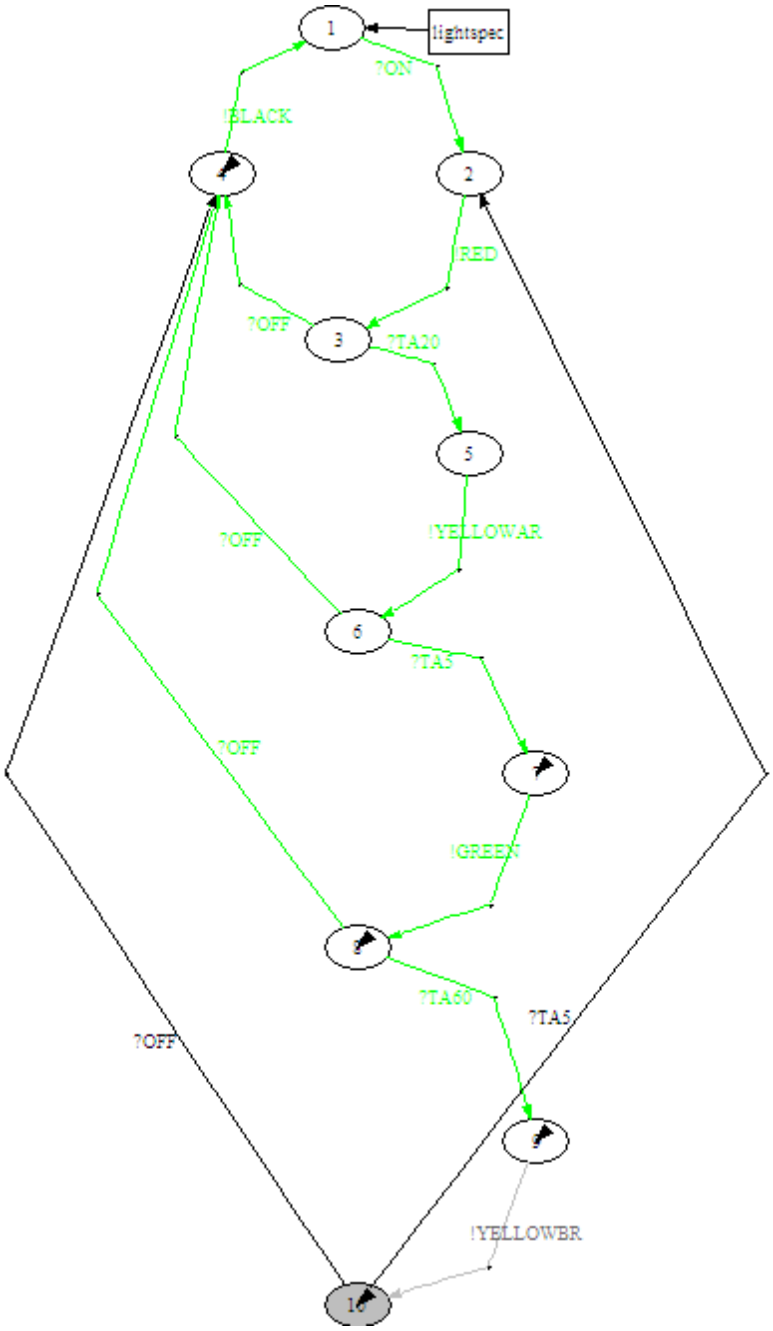


Fig. 20: Implementation 1 antidot model

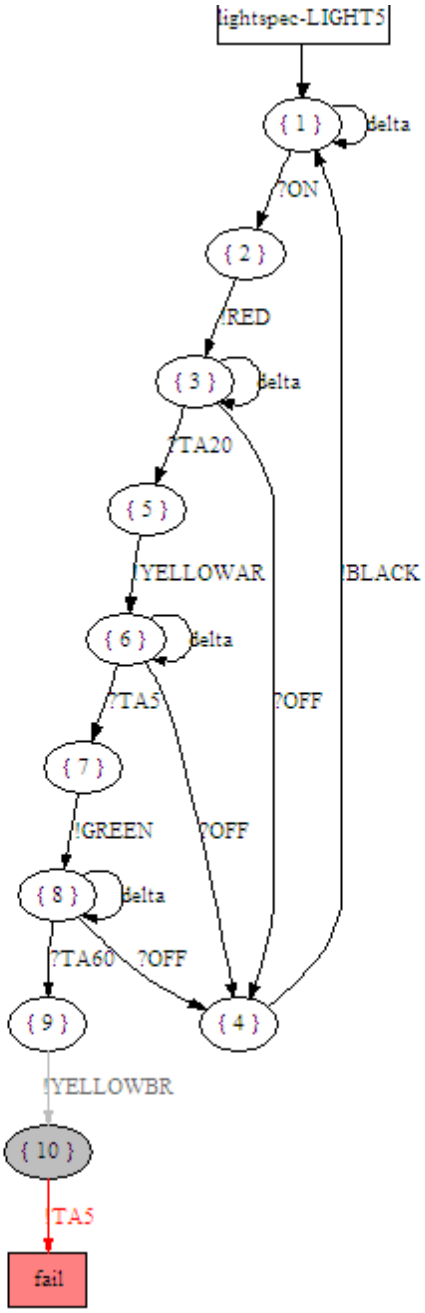


Fig. 21: Implementation 1 antidot test run

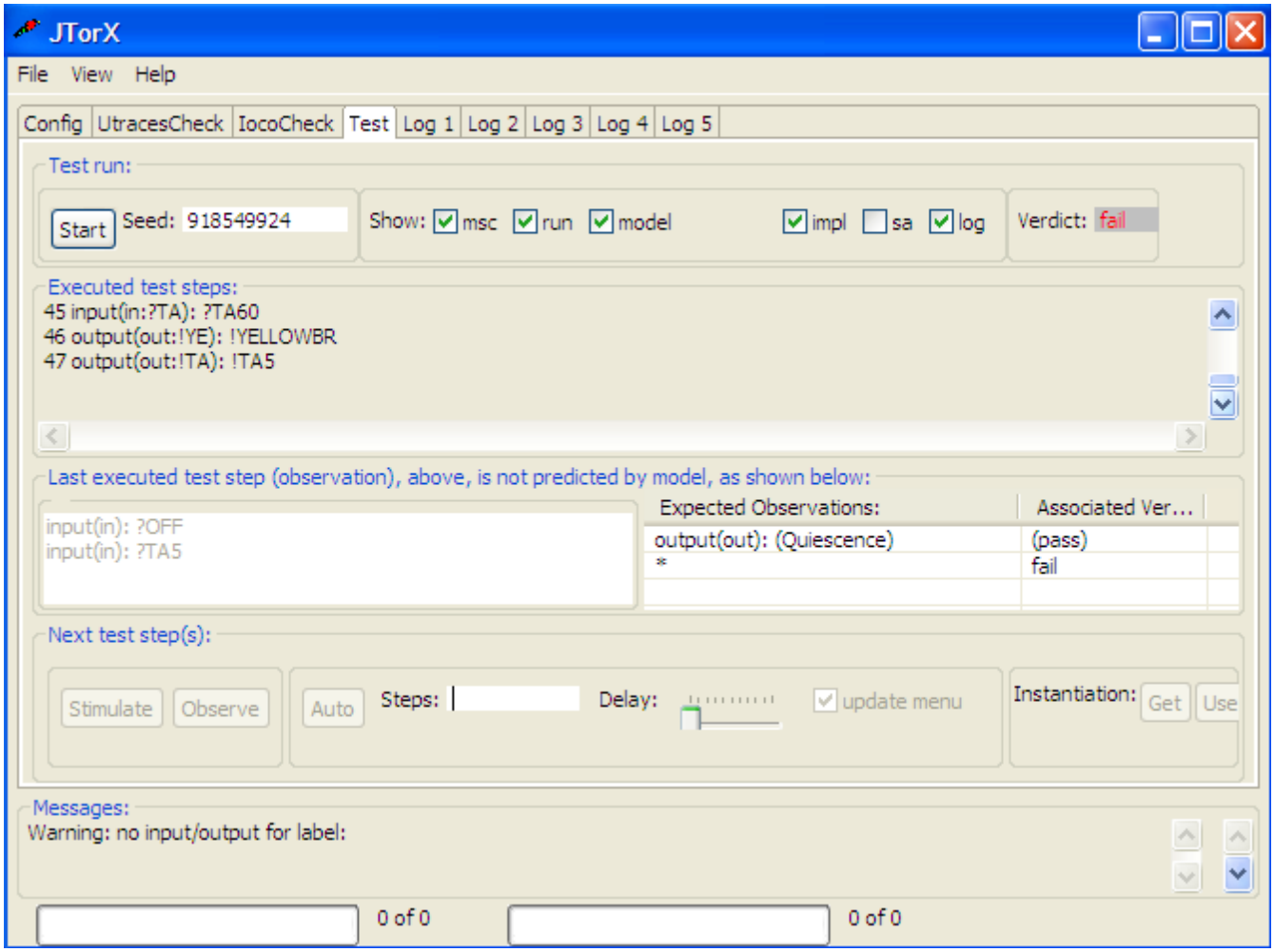


Fig. 22: Implementation 1 test result

Traffic Light tests: Implementation 2

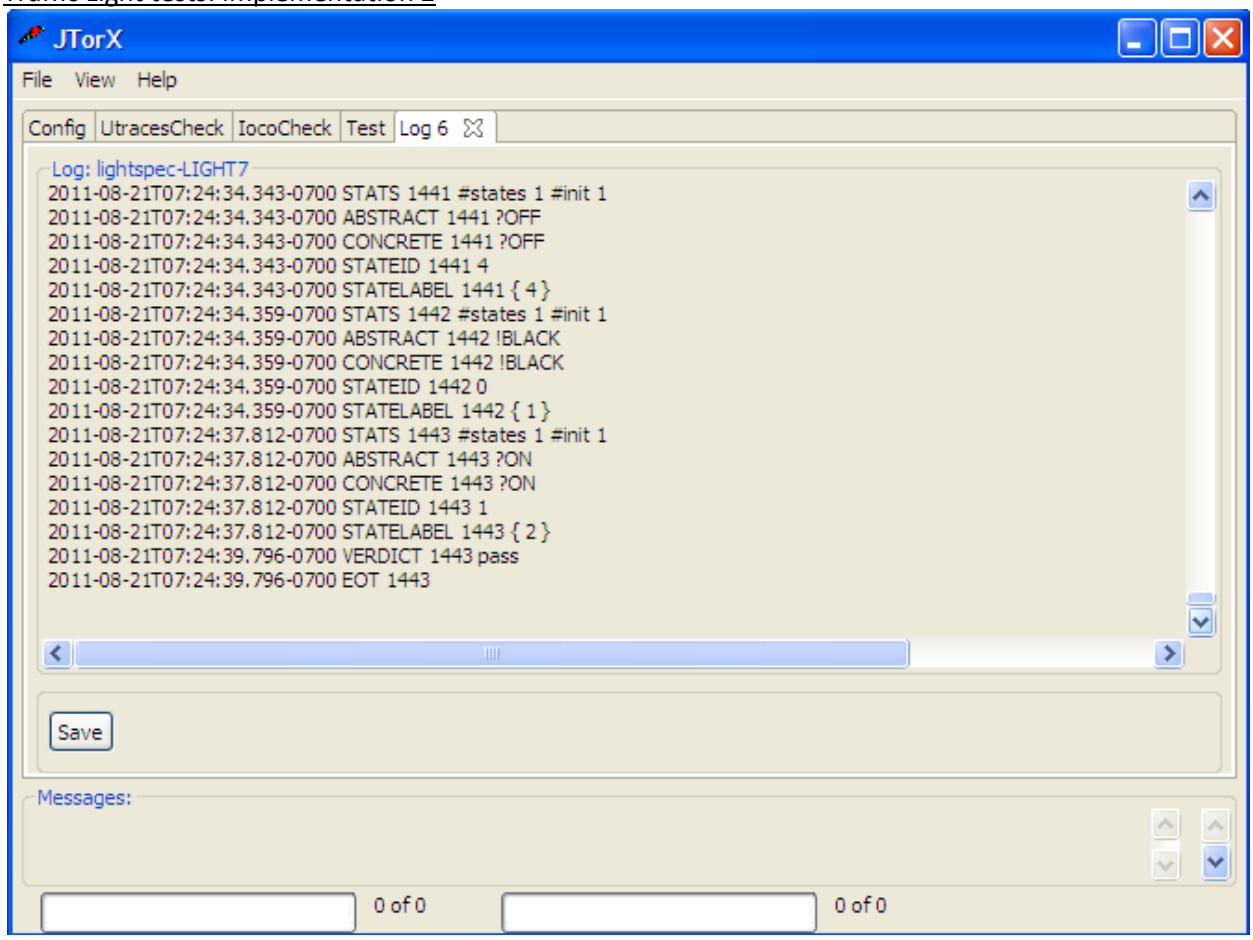


Fig. 23: Implementation 2 log

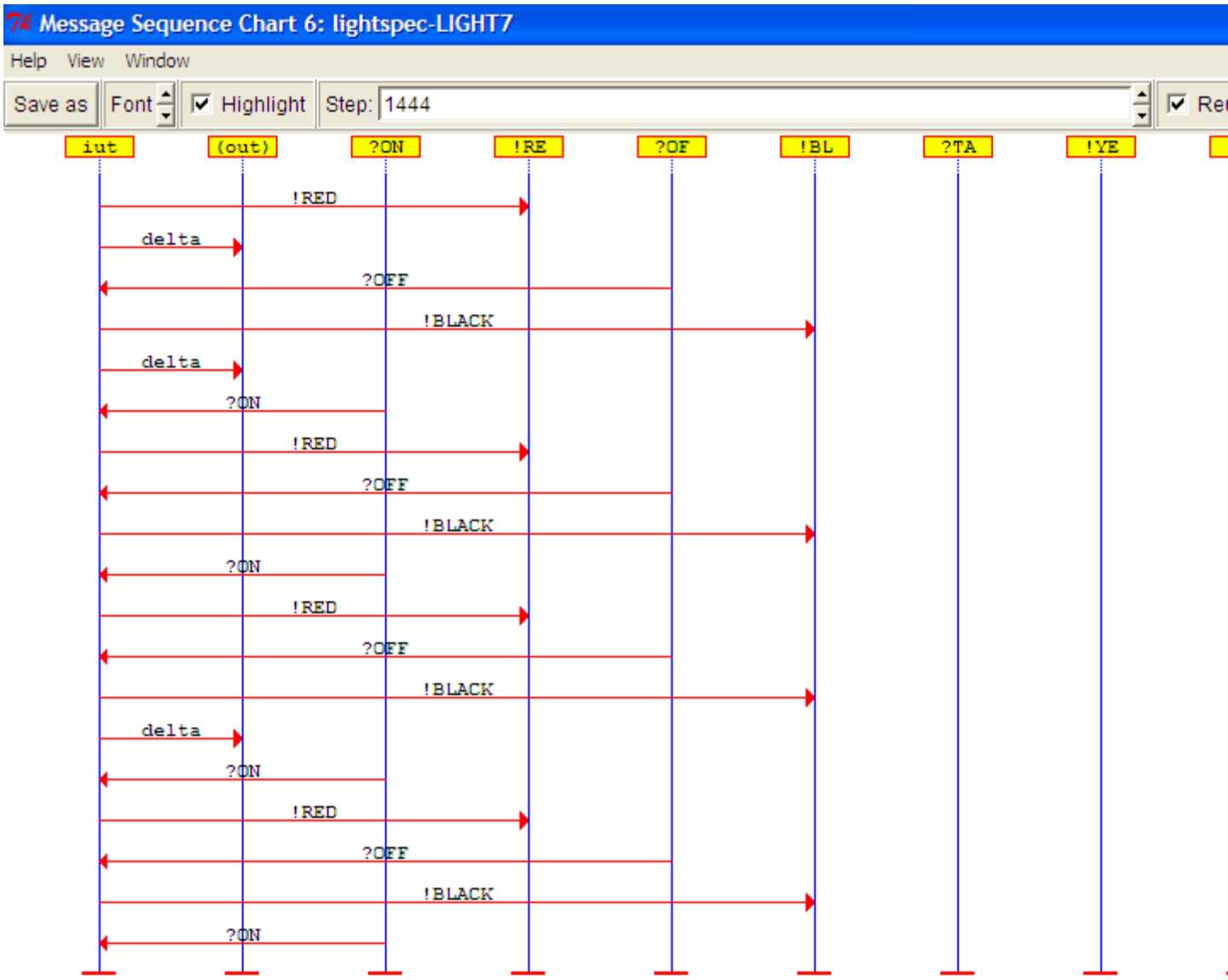


Fig. 24: Implementation 2 message sequence chart

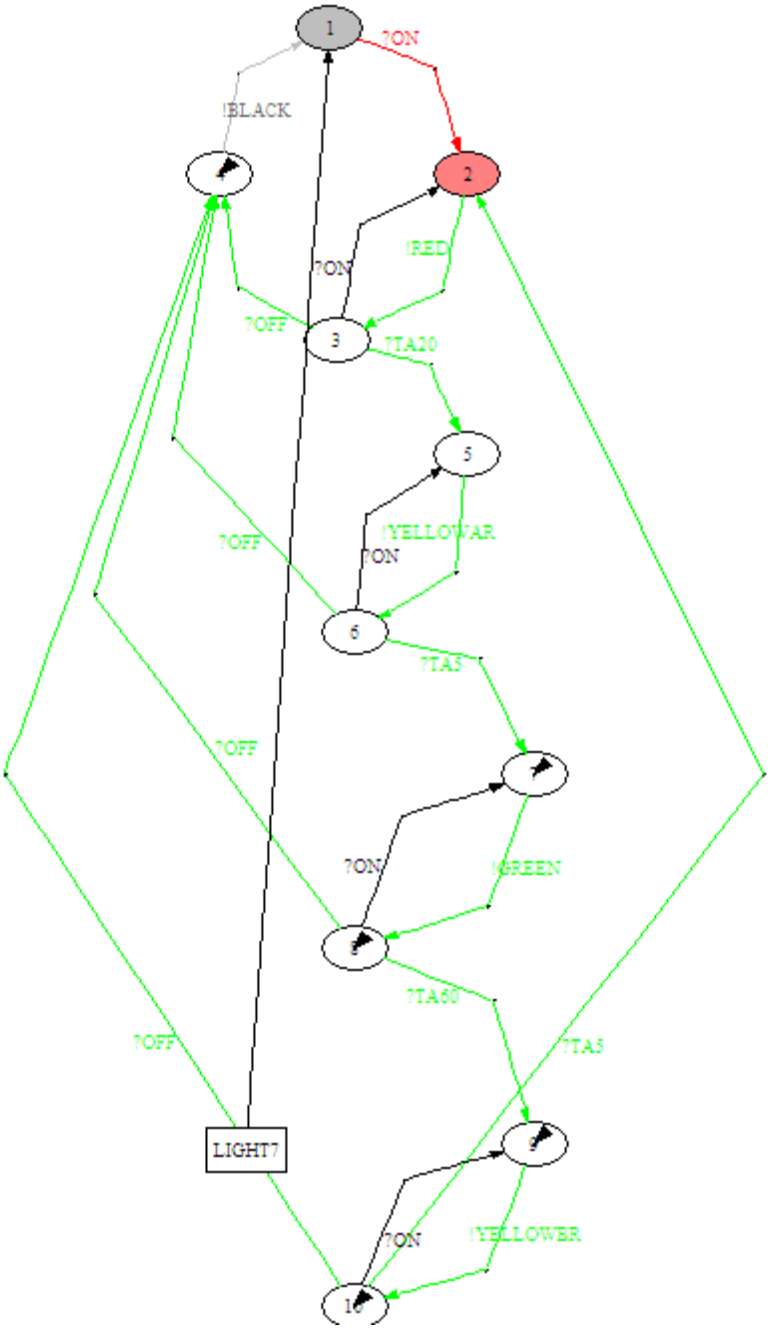


Fig. 25: Implementation 2 antidot implementation

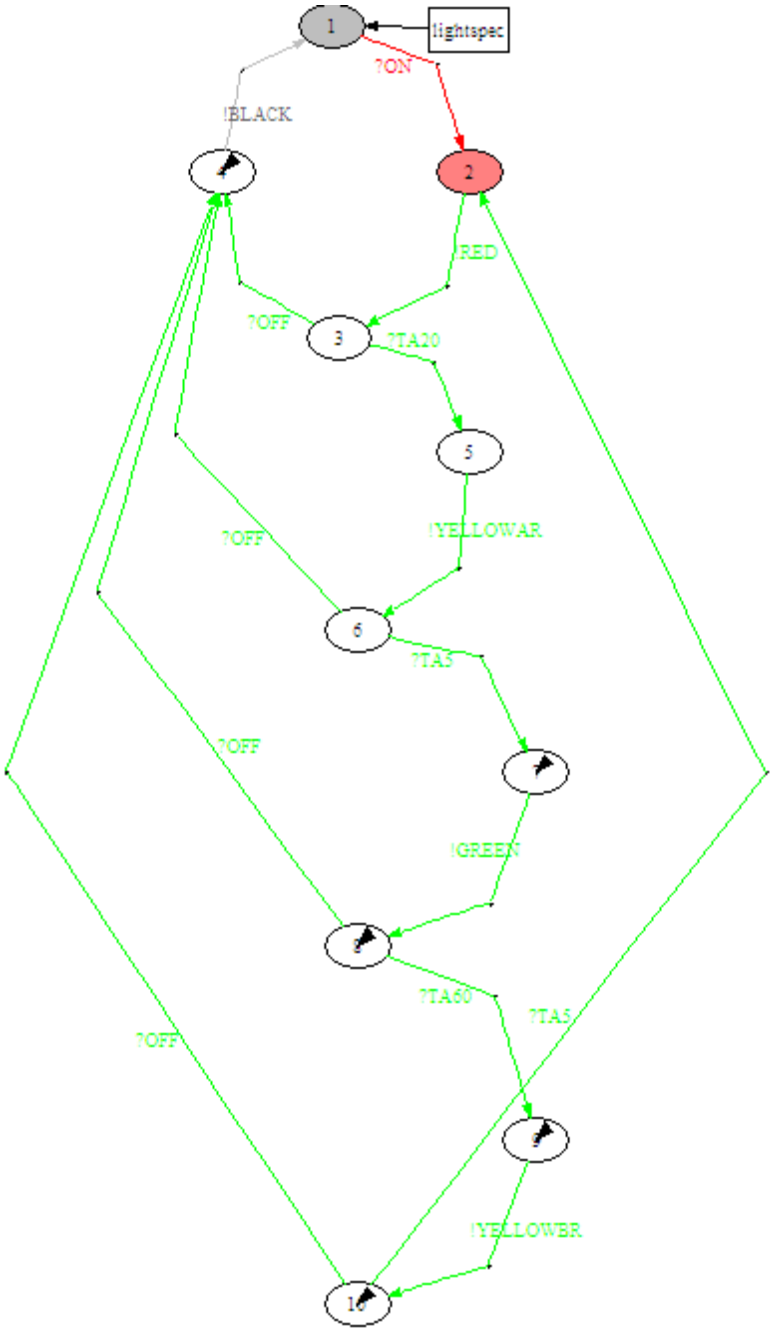


Fig. 26: Implementation 2 antidot model

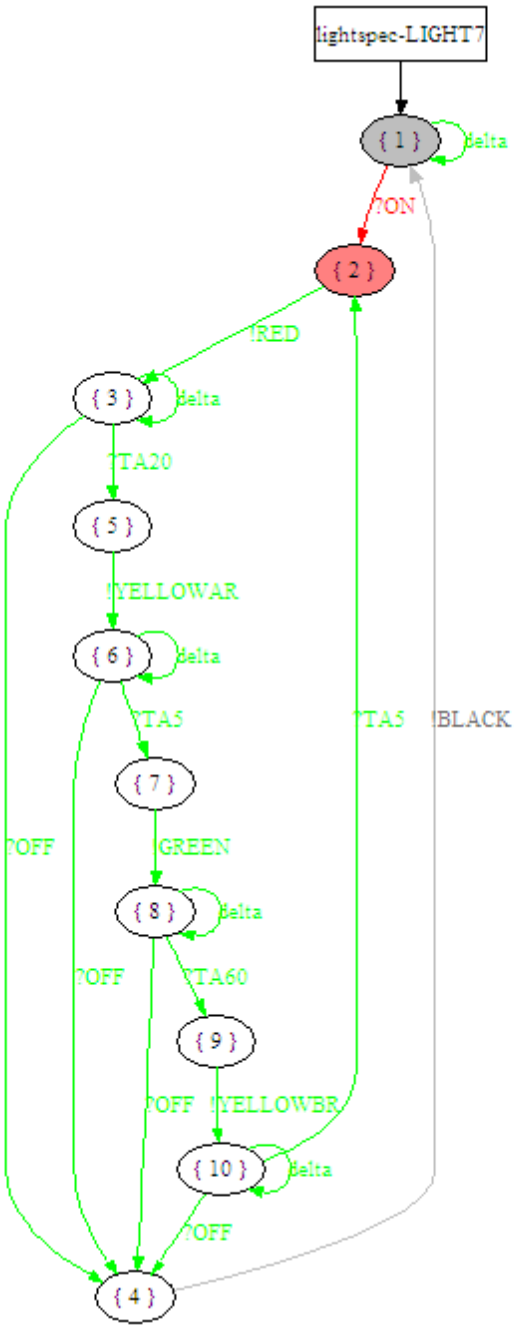


Fig. 27: Implementation 2 antidot test run

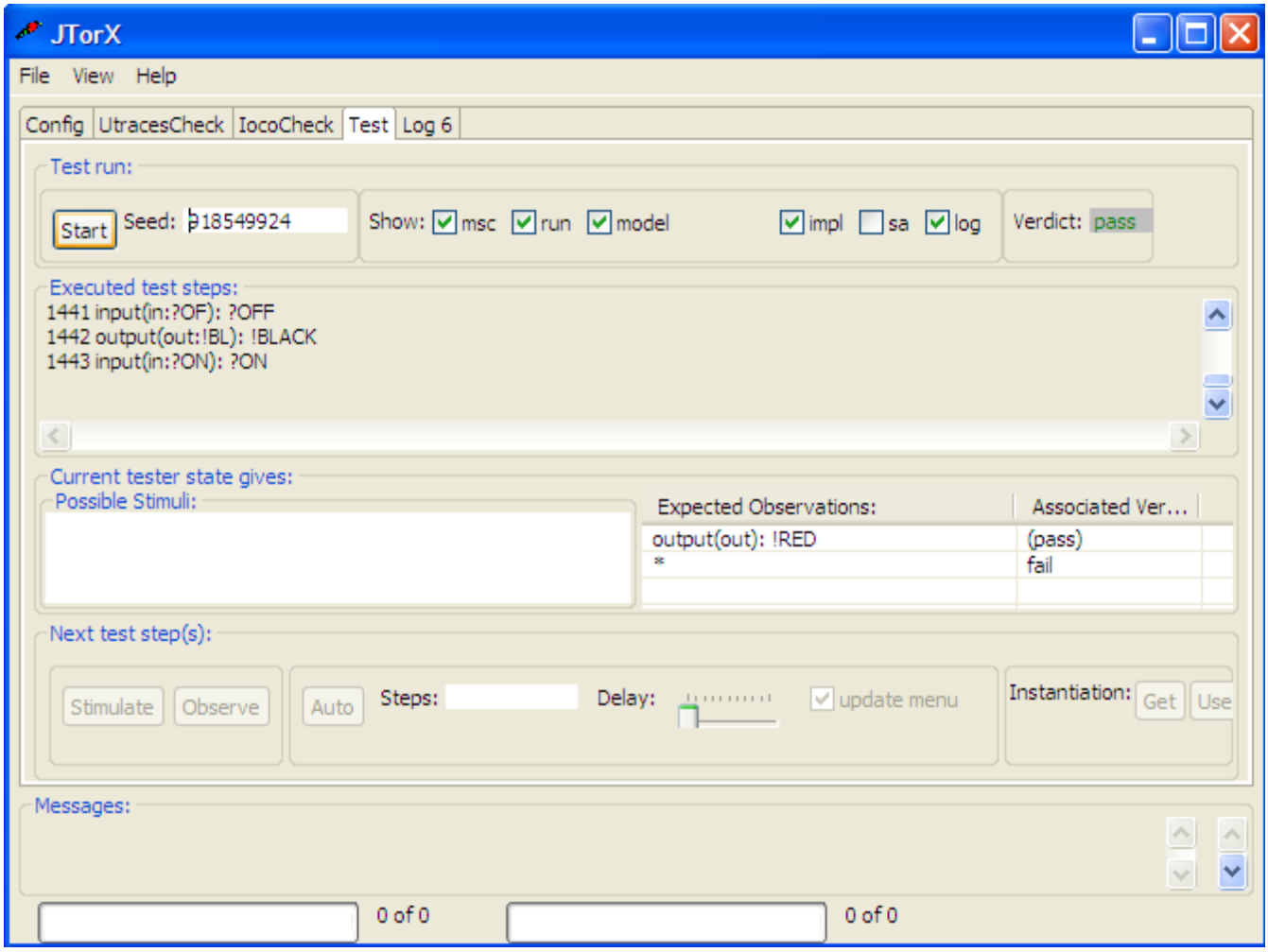


Fig. 28: Implementation 2 test result

Traffic Light tests: Implementation 3

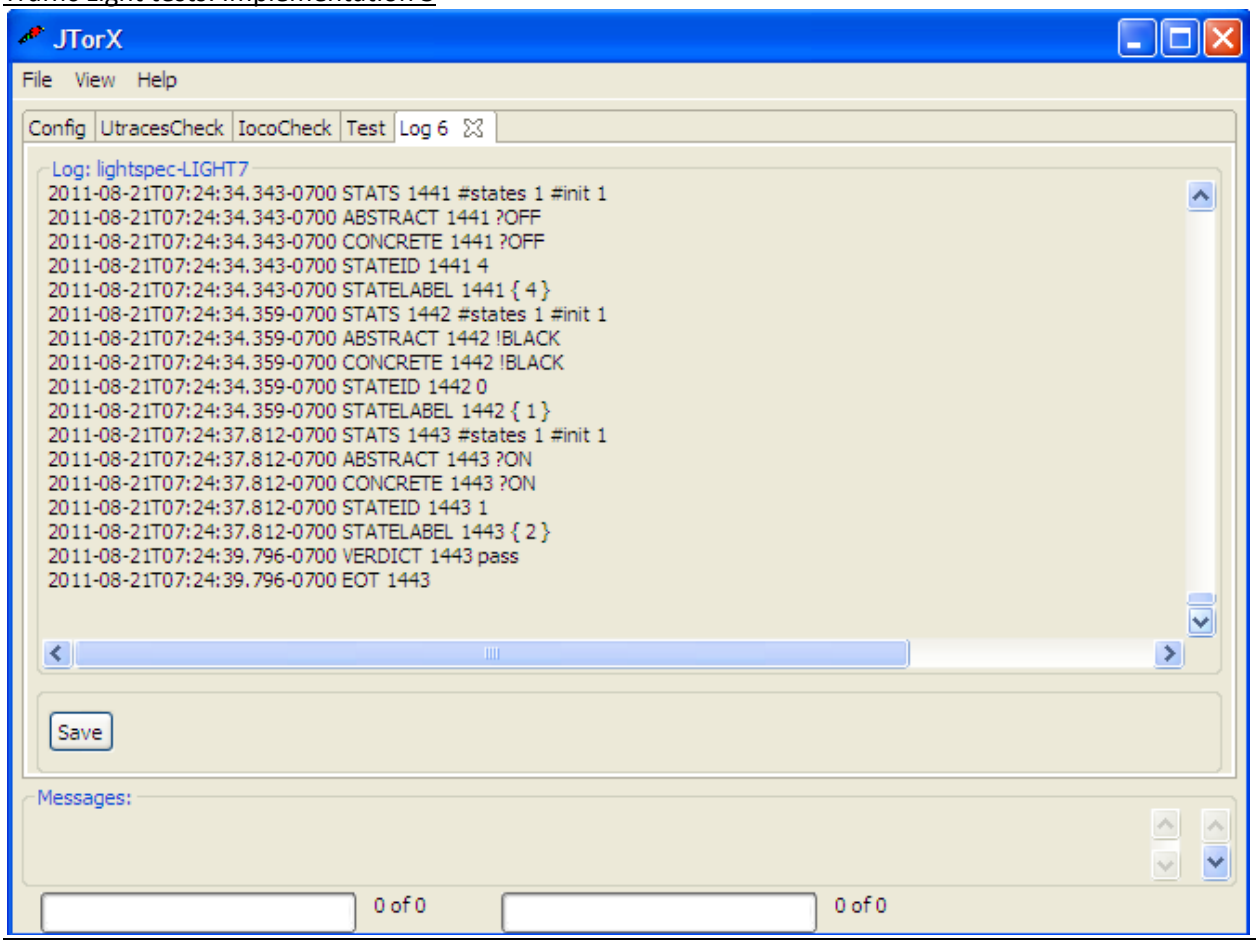


Fig. 29: Implementation 3 log

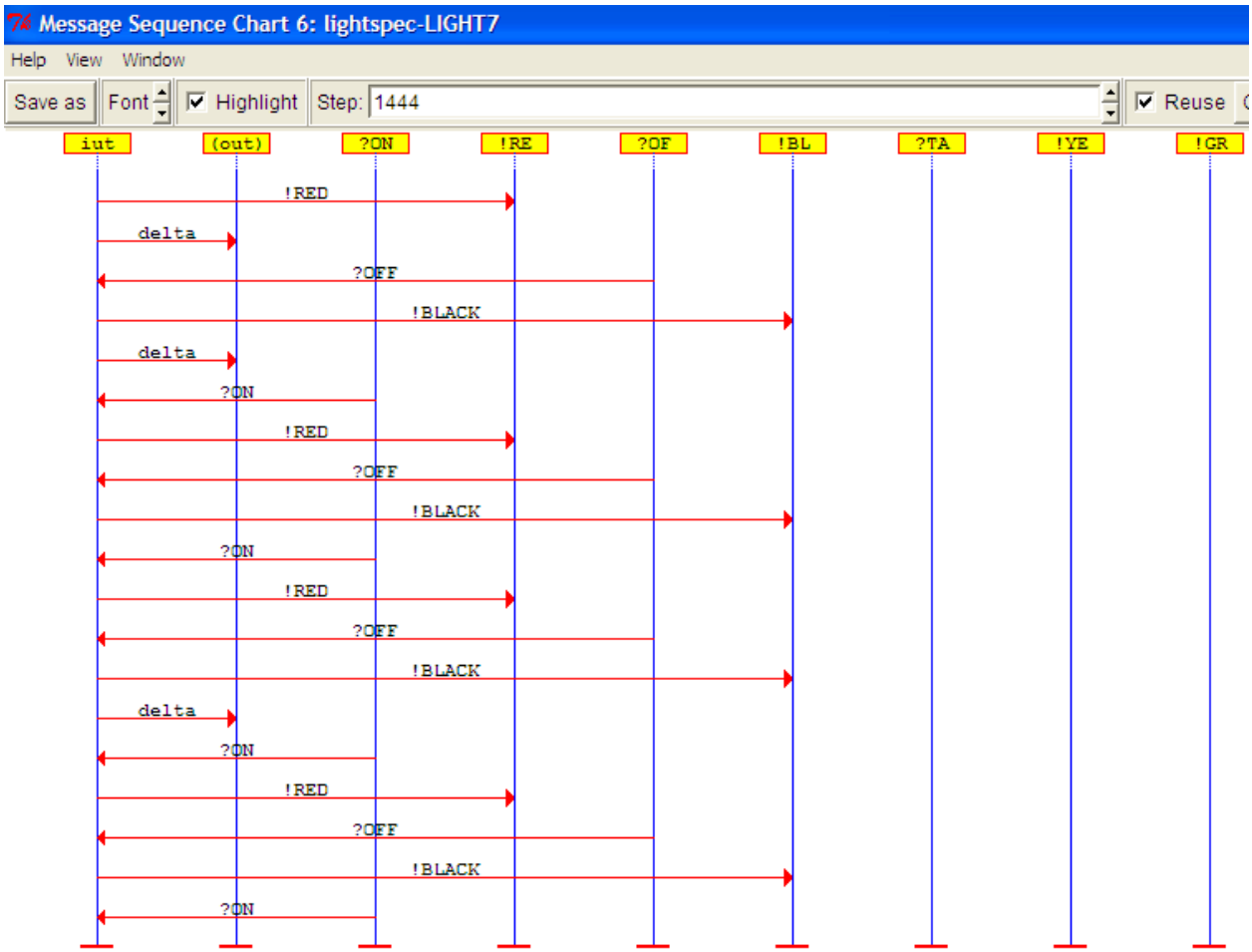


Fig. 30: Implementation 3 message sequence chart

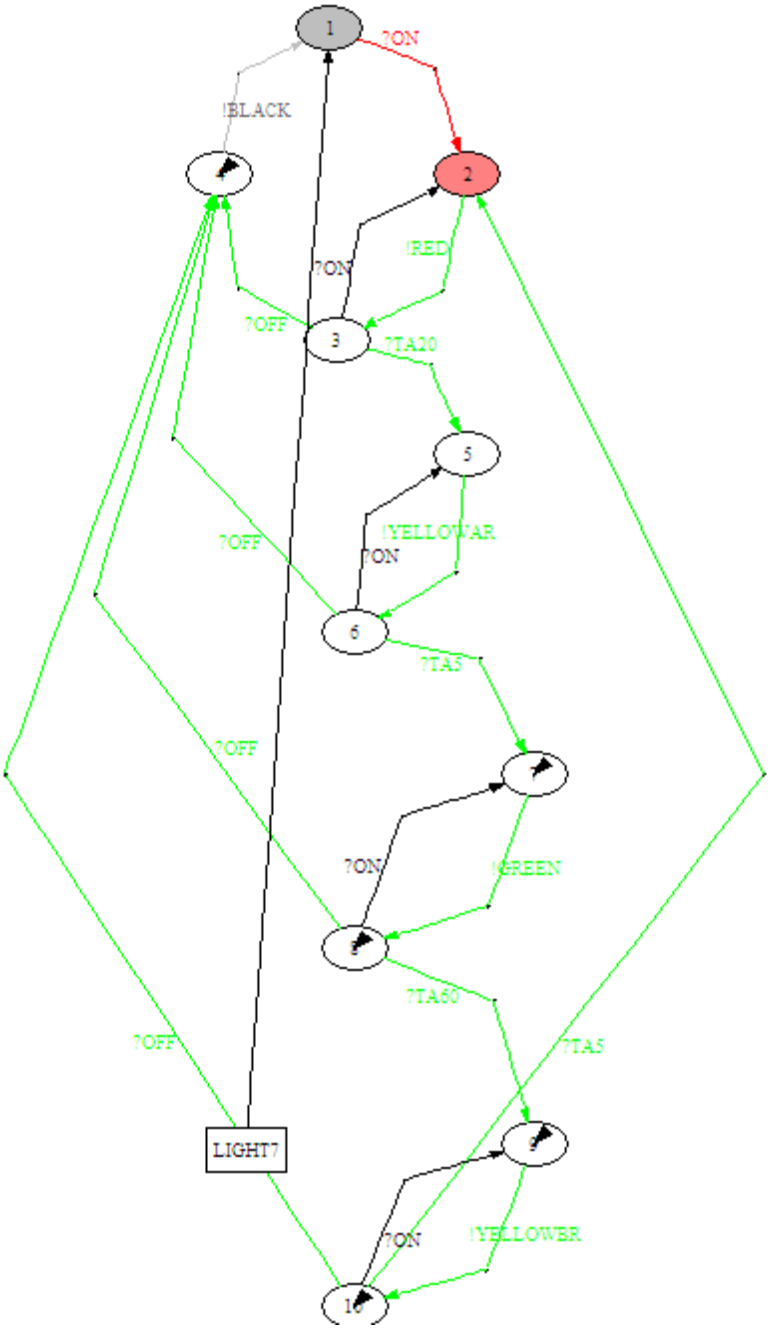


Fig. 31: Implementation 3 antidot implementation

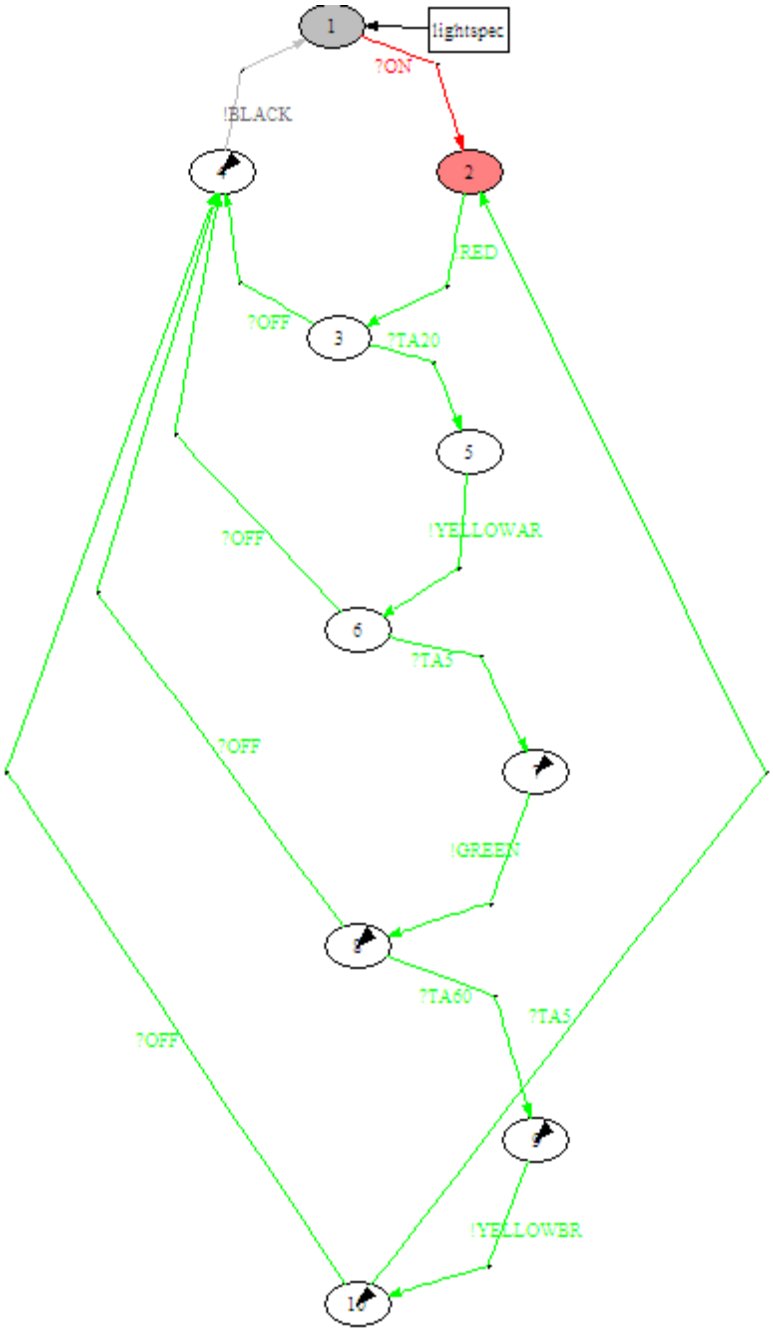


Fig. 32: Implementation 3 antidot model

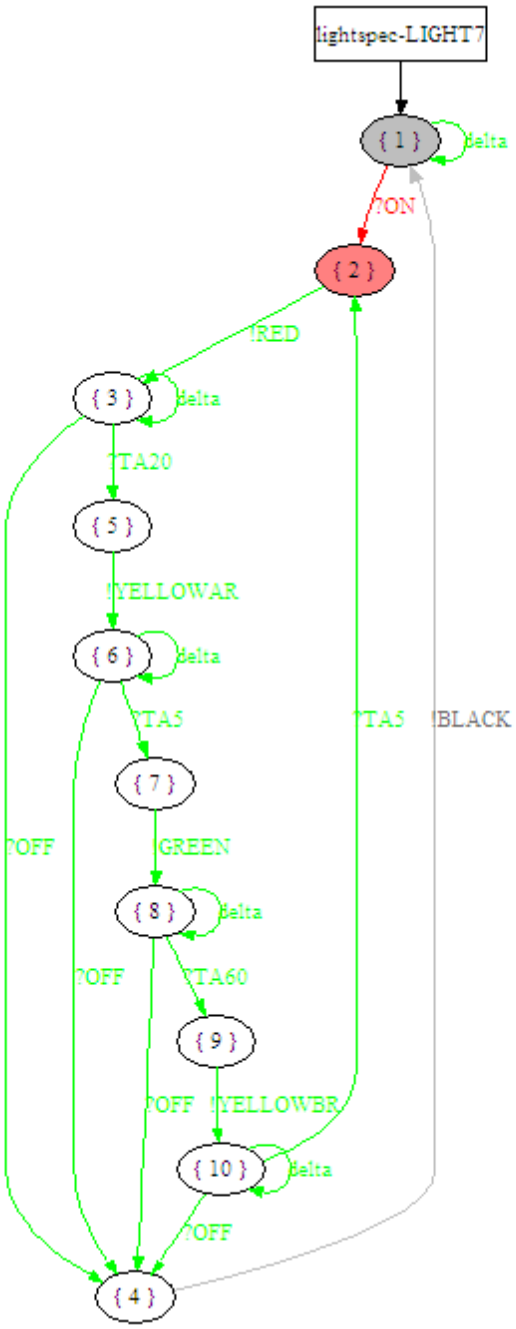


Fig. 33: Implementation 3 antidot test run

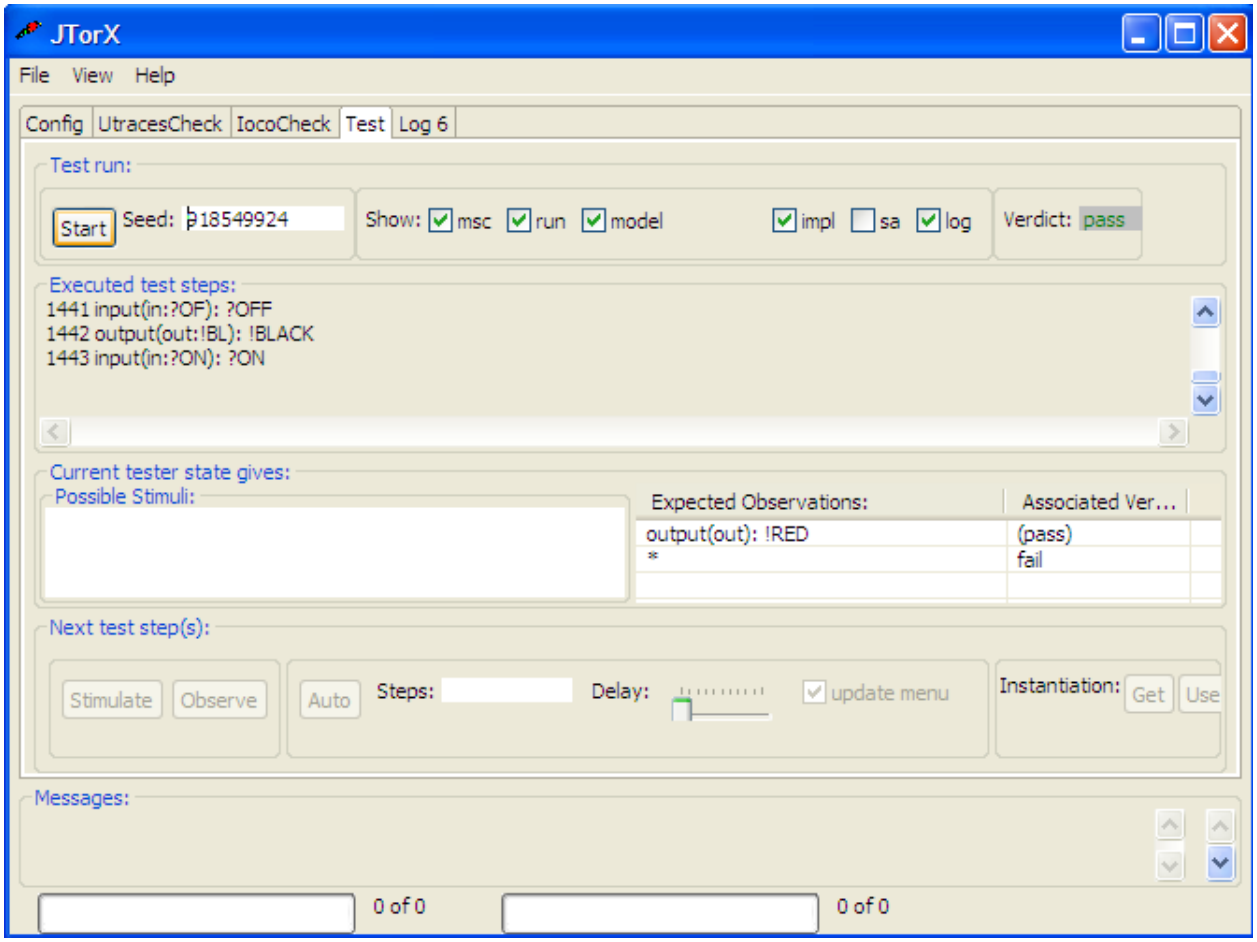


Fig. 34: Implementation 3 test result

Properties Examined

Input output conformance: From the implementation 1 run (figs 17-22), the non conformance result gotten shows that this implementation is wrong because it defines a new state for input Time advance 5 after Yellow AR rather than the state already defined in the specification. This confirms a non conformance implementation and shows that meanwhile, the LTS translation of the system specification conforms to its operational specification in implementations 2 and 3 (figs 22-27 and figs 28-34).

Completeness: The question of lack of completeness as raised in the LTSA runs is also revealed in the JTORX tests. Implementations 2 and 3 (figs 13 and 14) are different in the sense that they specify what happens when in a state and control on is inputted or control off are inputted at all states. This specification is lacking in the system specification as shown in the system specification in figure 11. And implementations 2 and 3 (figs 13 and 14) proves the correctness by comparing the corrected complete implementation in implementations 2 and 3 against the system specification one can see that the corrected (complete specification) conform with the specification as shown in the ioco check verdict of JTORX in figures 28 and 34 respectively.

Precedence property: Conformance to precedence property is shown in the correct implementations 2 and 3 (figs 13 and 14) and the proof that the specification explicitly specifies precedence is shown in implementation 1 (fig 12), where a wrong precedence in terms of the state after yellow BR with input time advance [5] shows that the verdict for ioco conformance because of this is fail in figures 21 and 22, and this is also seen in the message sequence chart and log in figures 17 and 18.

Eventuality: One can see from the message sequence chart of the correct implementations (implementations 2 and 3) as shown in figures 13 and 14 respectively and in their antidot animated model diagrams (figures 26 and 32) that eventually all states are reached and no state is left unreached. The presence of transition in all states from this model diagrams shows the proof of eventuality.

DDML IOS Traffic Light Summary

In summary, using the formal tools available at this level (JTORX and LTSA) to analyze the DDML IOS model through runs on the traffic light system. We have been able to test and check whether the operational specification of the DDML IOS conforms to the model, and that the model conforms to some property specification. From the test runs above, especially referring to ioco conformance tests by JTORX and the operational specification test run using FSP in LTSA, one can conclude from the results of the test runs as discussed above that the DDML IOS operational specification conforms to the models.

Furthermore, as shown in the test results above, some of the property specifications like progress and safety are in conformance as shown by test runs performed with these formal tools. These tools have been able to show lack of errors and deadlocks in the models, as well as reveal the certainty of progress in the model.

However, test runs from these tools also reveal weakness in some property specification conformance. Properties like completeness, precision and uniqueness are shown to be weak or lacking in the conformance test runs. Thus, properties like completeness, concision/precision and uniqueness would need to be revisited and addressed in the operational and property specification of DDML IOS.

6.2 DDML IORO Formal Analysis

The focus of formal analysis is the DDML IORO level. This level is concerned with DDML trajectory level that provides a footprint for the functional transition among states in a sequential manner. The DDML IORO model traces are mapped to temporal logic formal languages such as: Linear Temporal Logic (LTL) and Computational Tree Logic (CTL), to help assess the formal properties of the tool and efficiently analyze them.

Tools

The tool used in this work is the LTSA (labeled transition systems analyzer). LTSA is formal tool developed by Jeff Kramer and Jeff Magee at the Imperial College London. This tool is designed to help modelers formally analyze LTS models (labeled transition systems) written in FSP (finite state processes). However with the aid of the Fluent LTL specification of the LTSA, some LTL related properties such as mutual exclusion, safety properties, deadlocks and progress are checked.

In the course of this analysis, the traffic light case study is used in the performing this task. The traffic light case study is expected to model the road traffic light system and the IORO model diagram as well as the LTS translation is shown below (fig. 35 & 36).

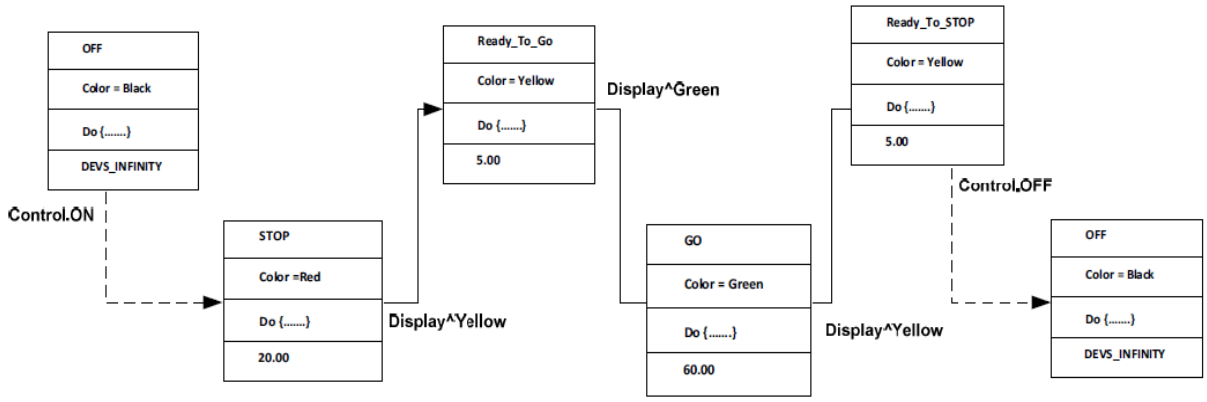


Fig 35: Traffic Light DDML IORO diagram

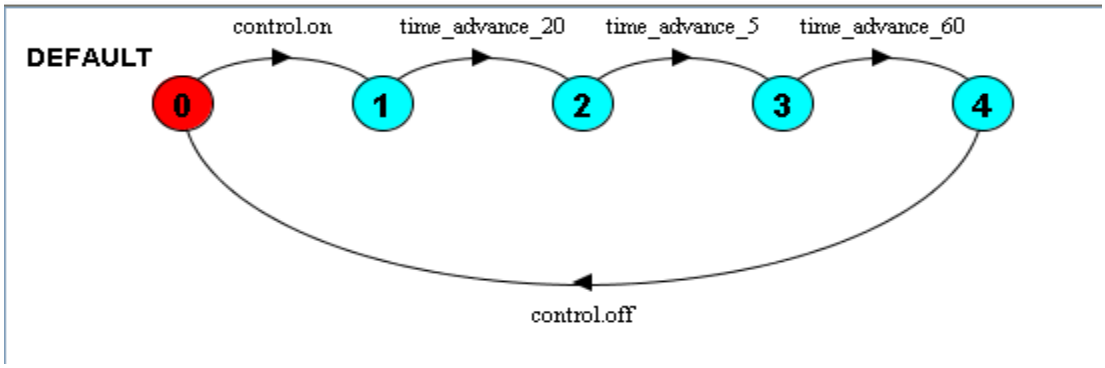


Fig 36: Traffic Light DDML IORO LTS diagram

LTSA tests

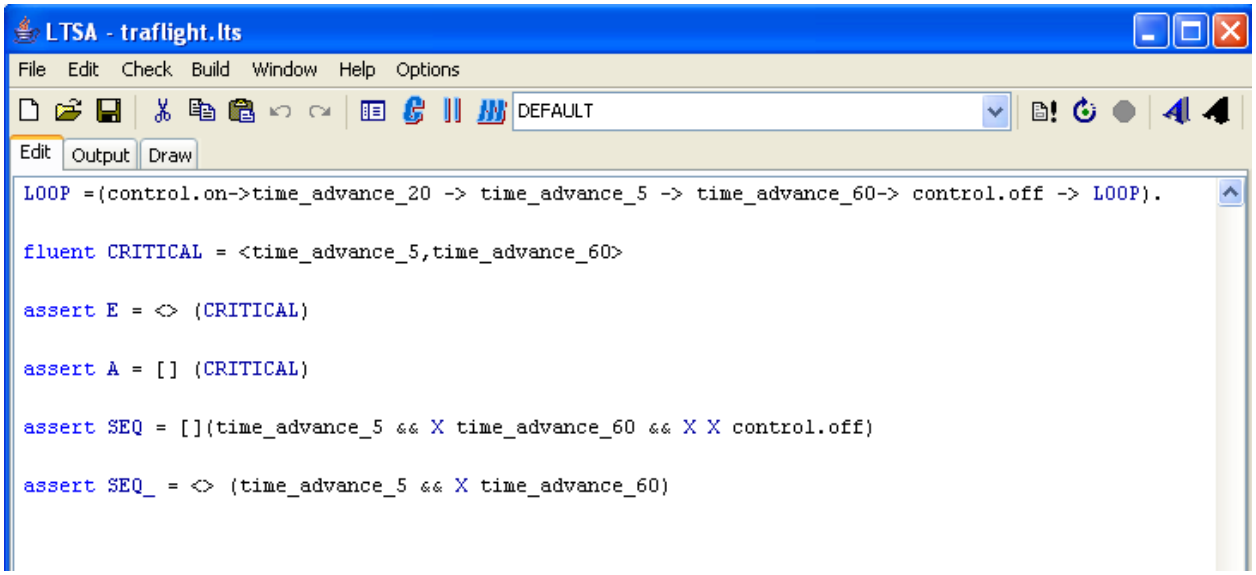


Fig 37: FSP code for the traffic Light system showing the FLTL specification and properties check

Property check:

The specification above in figure 8 shows the traffic light DDML IORO specification written in FSP. The code defines the fluent LTL titled CRITICAL (defined as the time advance 5 and time advance 60 activity sequence) as the critical section in line 2, showing the transient section of the state transition sequence as shown in the DDML IORO traffic light in figure 35, which in turn translates to the sequence: READY_TO_GO, GO and READY_TO_STOP. This sequence provides the transient part of the traffic light DDML IORO model and also interprets as the most critical part of the sequence which should not be interrupted by any external input stimuli. Therefore, the subsequent specification in the FSP code in figure 37 asserts that:

1. the critical section eventually (<>) occurs as defined in line 4 starting with: “assert E”
2. the critical section always ([]) occur i.e. this section is reachable in line 3 starting with: “assert A”
3. the sequence (time advance 5, time advance 60 and control off) always ([]) occur and

4. the sequence of the critical section (time advance 5 and time advance 60) eventually (<>) occur

In summary, we are checking for the eventuality and repeated occurrence properties of the critical section (properties 1 & 2) in the FSM graph and the precedence properties as shown in the sequence definition in properties 3 and 4. Furthermore, to confirm these afore-mentioned properties, the safety, progress and precedence properties of the traffic light specification is checked and the results showing a success verdict is shown in figures 38, 39 and 40.

Results:

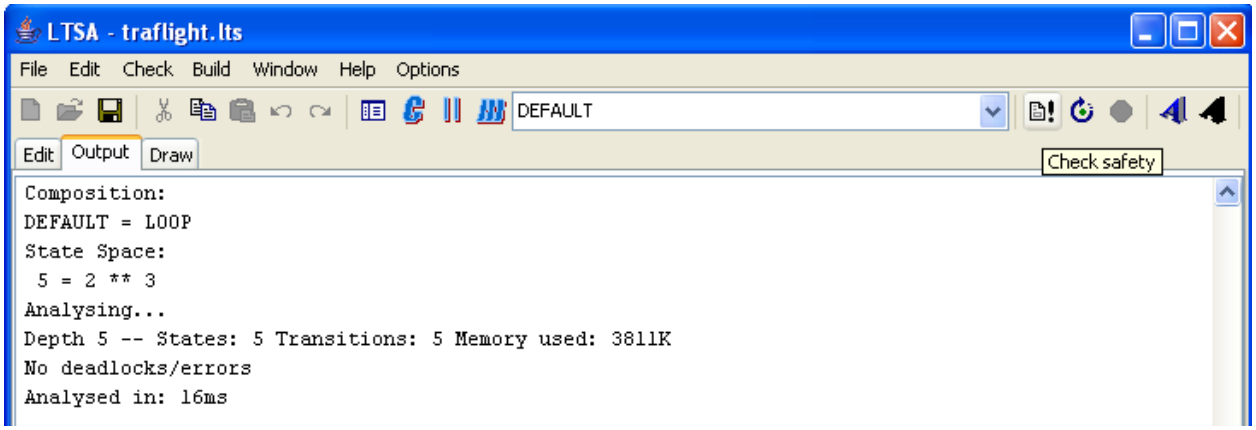


Fig 38: traffic Light DDML IORO Safety check

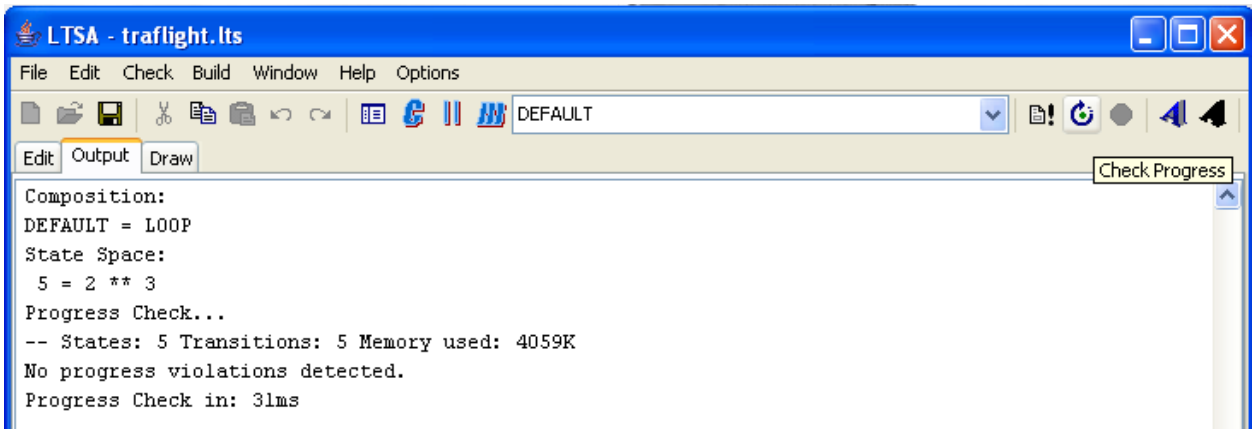


Fig 39: traffic Light IORO Progress Check

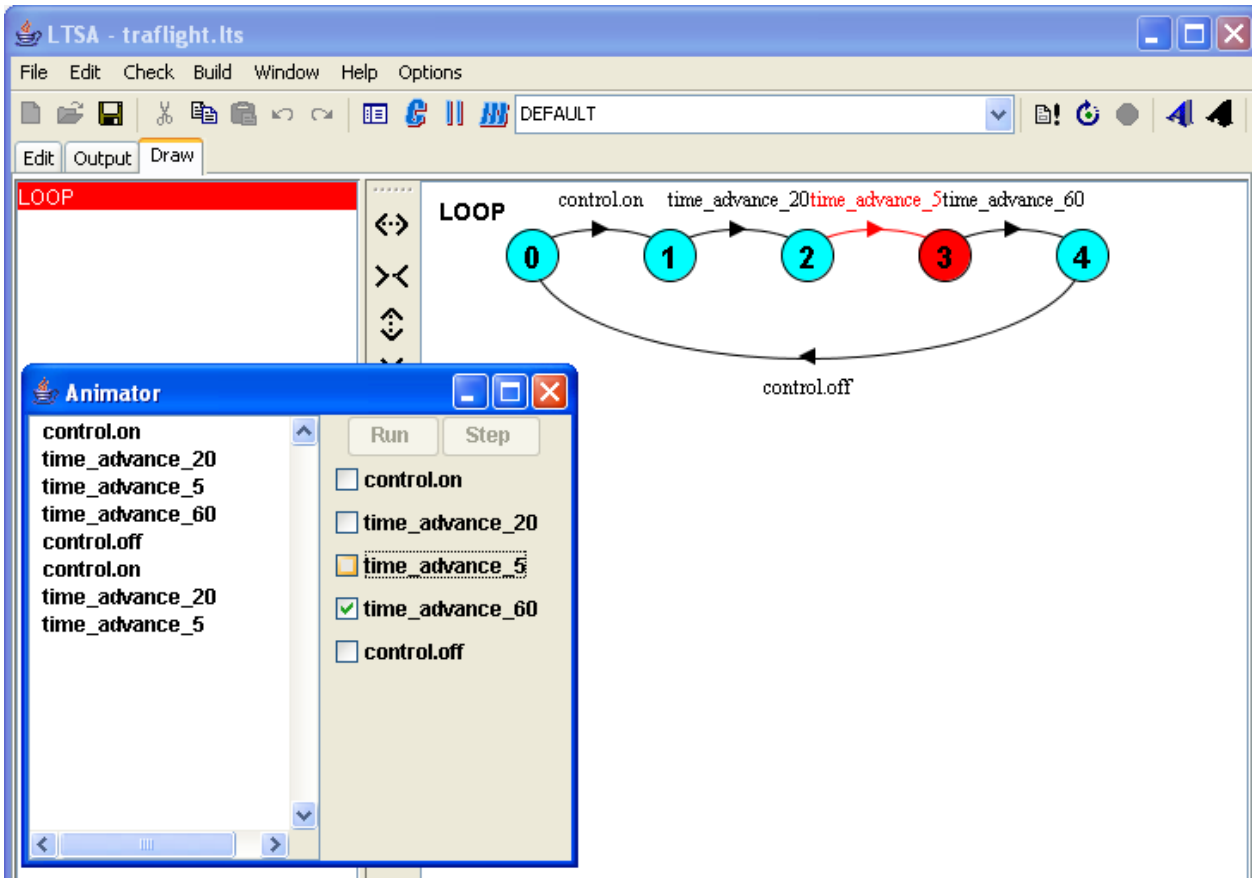


Fig 40: traffic Light IORO Sequence (precedence property) check

DDML IORO LTSA test Summary

In summary, using the LTSA formal tool to analyze the DDML IORO model through test runs performed on the traffic light system modeling case studies, we have been able to test and check whether the operational specification of the DDML IORO conforms to the model, and that the model conforms to some property specification. From the test runs above, referring to the operational specification test run using FSP in LTSA, one can conclude from the results of the test runs as discussed above that the DDML IORO operational specification conform to their models.

Furthermore, as shown in the test results above, some of the property specifications like precedence, eventuality, progress and safety are in conformance as shown by test runs performed with this tool. The LTSA tool has been able to show lack of errors and deadlocks in the models, as well as reveal the certainty of progress and safety in the model.

6.3 DDML CN test using Process Analysis Toolkit (PAT)

Using process analysis toolkit (PAT) we test the DDML CN level of the Traffic light system with the coupled network of the roads and the platforms in between the roads, in which cases we would be checking for interleaving without barrier synchronization with the aid of the interleaving operator (|||)

in CSP. This operator defines that processes can run simultaneously but no event or activity is synchronized. That is no two events similar to both processes are performed at the same time, unlike in the case of the parallel composition operator (||) that allows lock step synchronization, two events performed simultaneously.

For example given:

1. P || Q
2. P ||| Q

And P is defined as a->c-> Stop and Q is defined c-> Stop, the first definition (1) above states that q waits for P to finish event a before it starts so that both P and Q run event c simultaneously. Meanwhile the second definition states that P and Q can run simultaneously interleaving and should not run two events from both processes simultaneously.

Below is the DDML coupled network diagram of the traffic light system showing the interactions among the roads and platforms (figure 45), and the diagram describing the positioning of the traffic lights (fig 46).

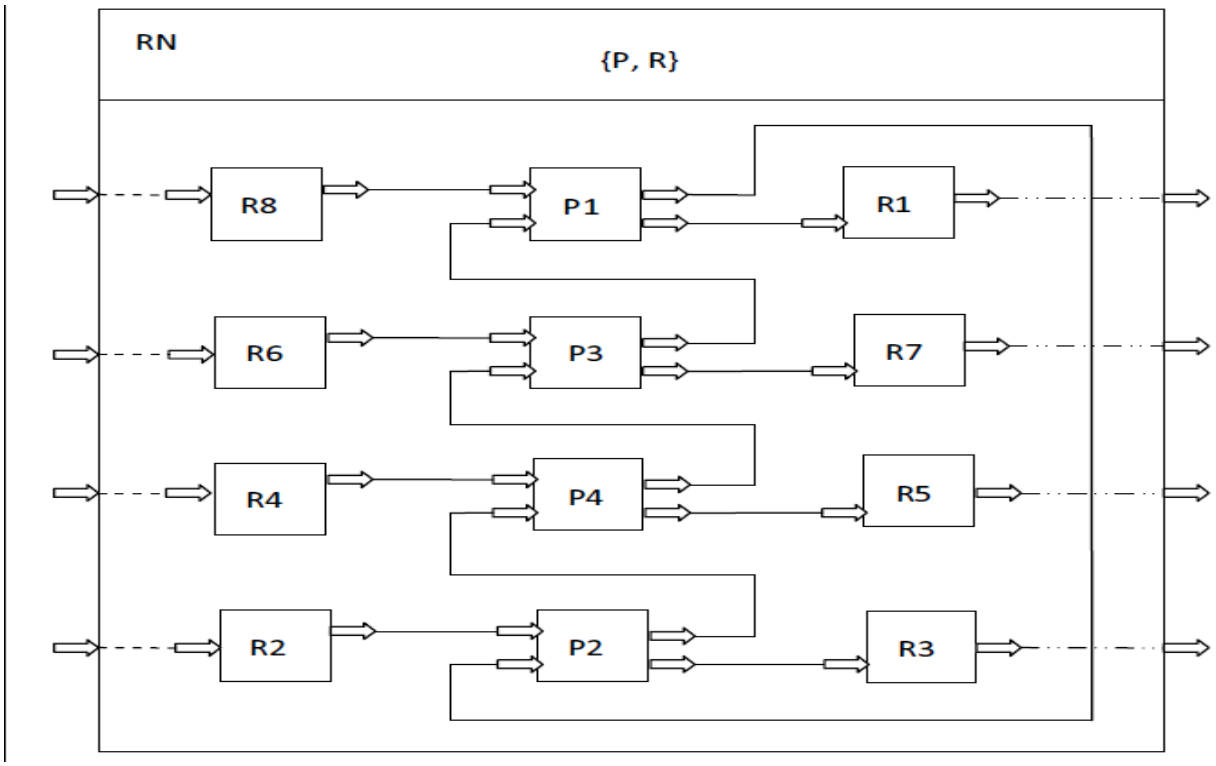


Fig 45: DDML CN level diagram of the traffic Light System, P represents platforms between roads and R represents the Roads.

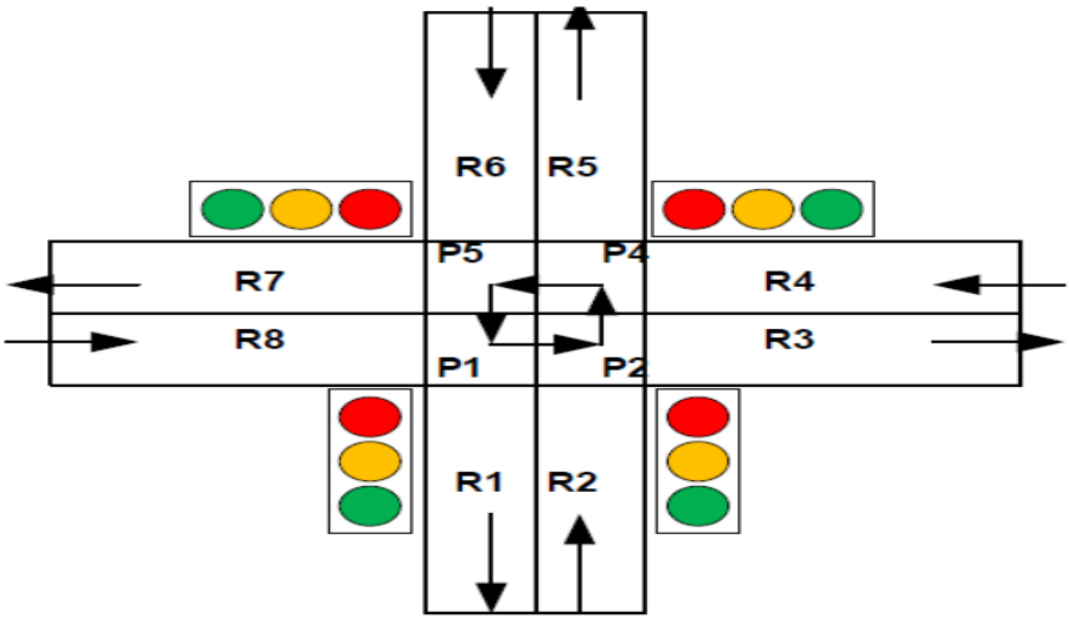


Fig 46: DDML CN level diagram of Traffic light system showing the positioning of the traffic lights, platforms and roads at the cross road.

From the above diagram in figures 45 and 46, we can see that there are 16 road routes for a car through the Cross road, moving form road to platform(s) till the car enters an exit road. These routes are expressed in CSP processes shown in the diagram of the CSP code written in PAT fig 47 below.

```
route1() = road2 -> p2 -> road3 -> route1();
route2() = road2 -> p2 -> p4 -> road5 -> route2();
route3() = road2 -> p2 -> p4 -> p5 -> road7 -> route3();
route4() = road2 -> p2 -> p4 -> p5 -> p1 -> road1 -> route4();

route5() = road4 -> p4 -> road5 -> route5();
route6() = road4 -> p4 -> p5 -> road7 -> route6();
route7() = road4 -> p4 -> p5 -> p1 -> road1 -> route7();
route8() = road4 -> p4 -> p5 -> p1 -> p2 -> road3 -> route8();

route9() = road6 -> p5 -> road7 -> route9();
route10() = road6 -> p5 -> p1 -> road1 -> route10();
route11() = road6 -> p5 -> p1 -> p2 -> road3 -> route11();
route12() = road6 -> p5 -> p1 -> p2 -> p4 -> road5 -> route12();

route13() = road8 -> p1 -> road1 -> route13();
route14() = road8 -> p1 -> p2 -> road3 -> route14();
route15() = road8 -> p1 -> p2 -> p4 -> road5 -> route15();
route16() = road8 -> p1 -> p2 -> p4 -> p5 -> road7 -> route16();


proces = route1()|||route2()|||route3()|||route4()|||route5()|||route6()|||route7()||

#assert proces deadlockfree;
#assert proces divergencefree;
```

Fig 47: CSP code for the DDML CN level showing the processes representing the routes from each road to through the platforms to every exit road.

However, this code has a large state space and it can not be verified, there exist a state explosion problem for this huge state set. The memory and time required simulating the state space and verifying divergence and deadlock freeness is high and can not be computed by PAT as shown in the diagram below in fig 48 revealing the inability to resolve the verification tests due to huge state space and out of memory error.

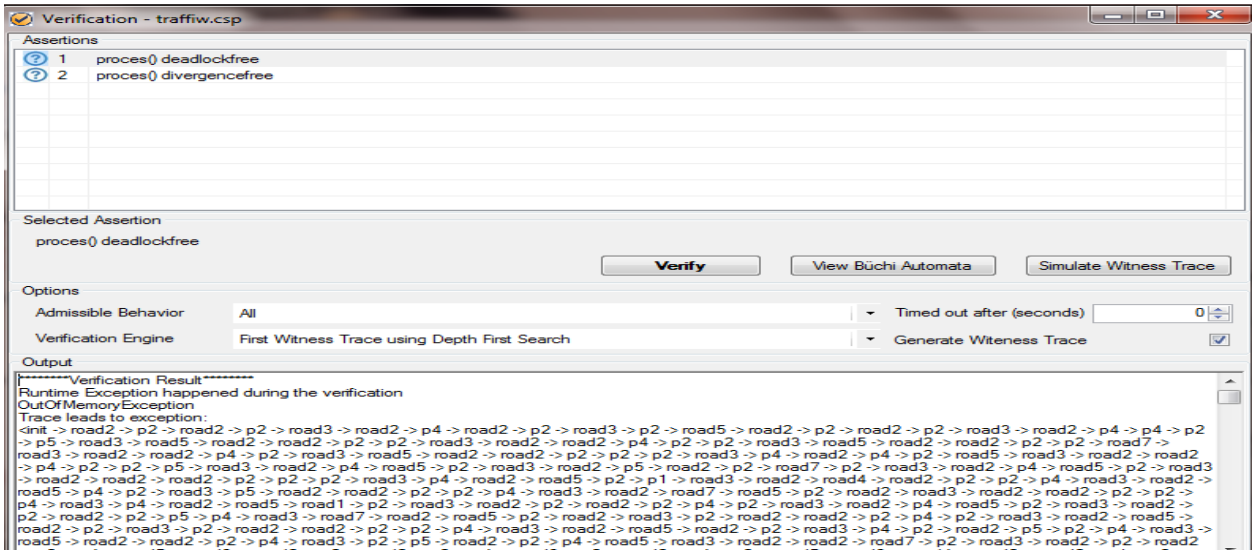


Fig 48: revealing the inability to resolve the verification tests due to huge state space and out of memory error. Runtime exception due to the state explosion problem

Therefore, we decided to solve the state explosion problem by reducing the state space to only state configuration clusters, strictly states that exhaust the platforms in all routes. So the routes were reduced to four and the following CSP code was written instead (fig 49).

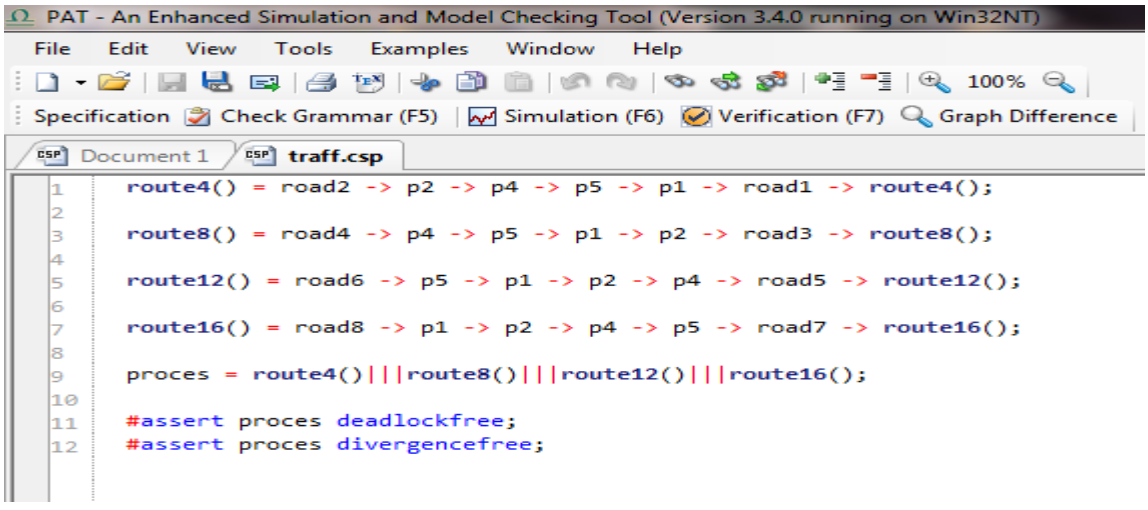


Fig 49: CSP code for the DDML CN level showing the configuration processes only, representing routes from each road to through the platforms to every exit road that exhausts the number of platforms.

DDML CN LEVEL CROSS ROAD ROUTE CONCURRENCEY TESTS

Output Window

```
Specification is parsed in 0.6056901s
//====================Process Definitions====================
route4()=
(road2->(p2->(p4->(p5->(p1->(road1->route4()))))));

route8()=
(road4->(p4->(p5->(p1->(p2->(road3->route8()))))));

route12()=
(road6->(p5->(p1->(p2->(p4->(road5->route12()))))));

route16()=
(road8->(p1->(p2->(p4->(p5->(road7->route16()))))));

proces()=
((route4())
|||(route8())
|||(route12())
|||(route16())
);



//============================Asserstion Definitions============================
#assert proces() deadlockfree;
#assert proces() divergencefree;
```

Grammar Checked traff.csp

Fig 50: Grammar check for cross road routes

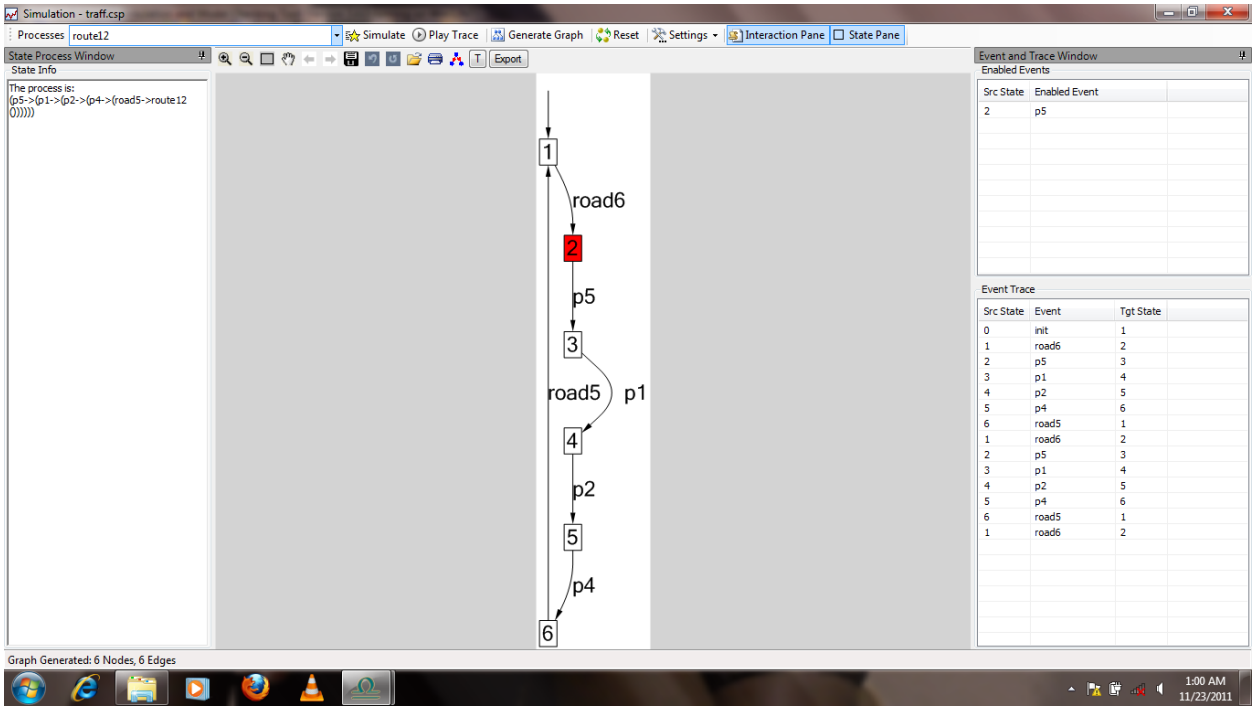


Fig 51: simulation of cross road route 12

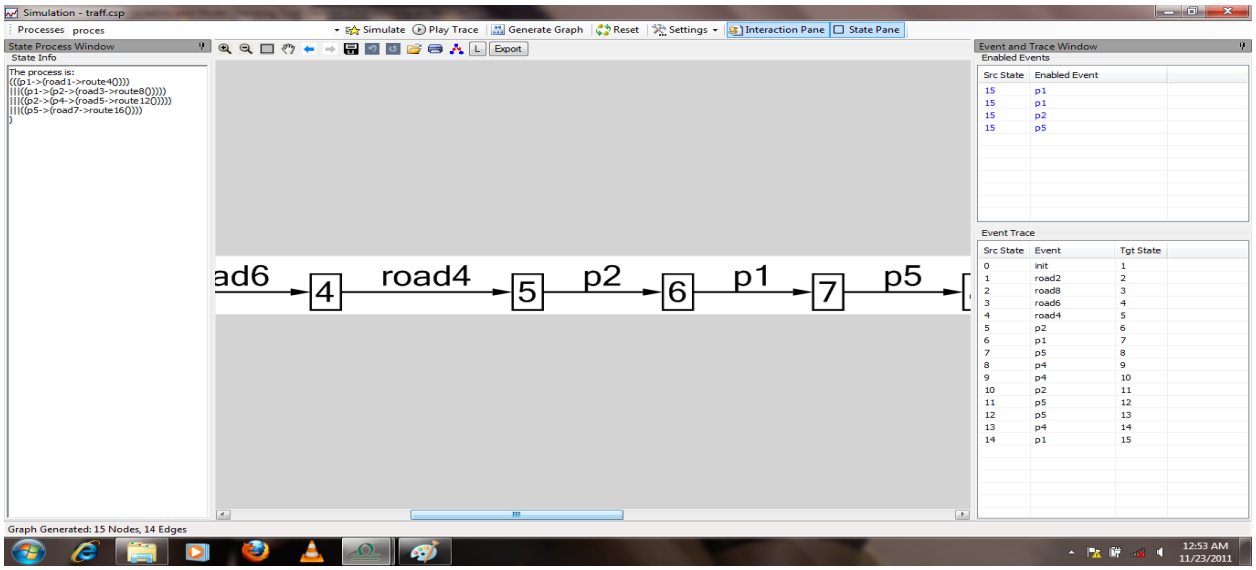


Fig 52: simulation of process of all four rotes

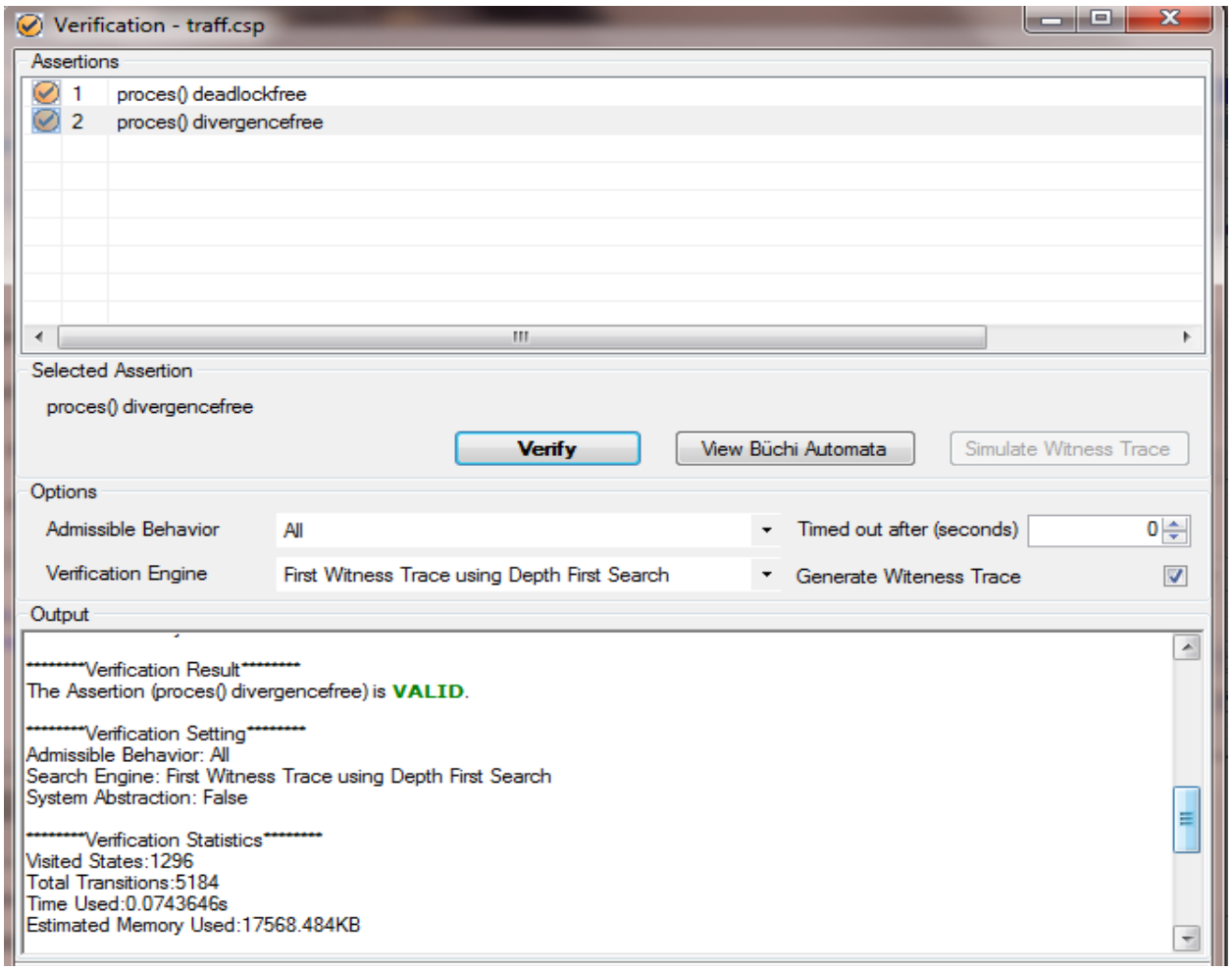


Fig 53: verification of divergence freeness for the crossroad routes

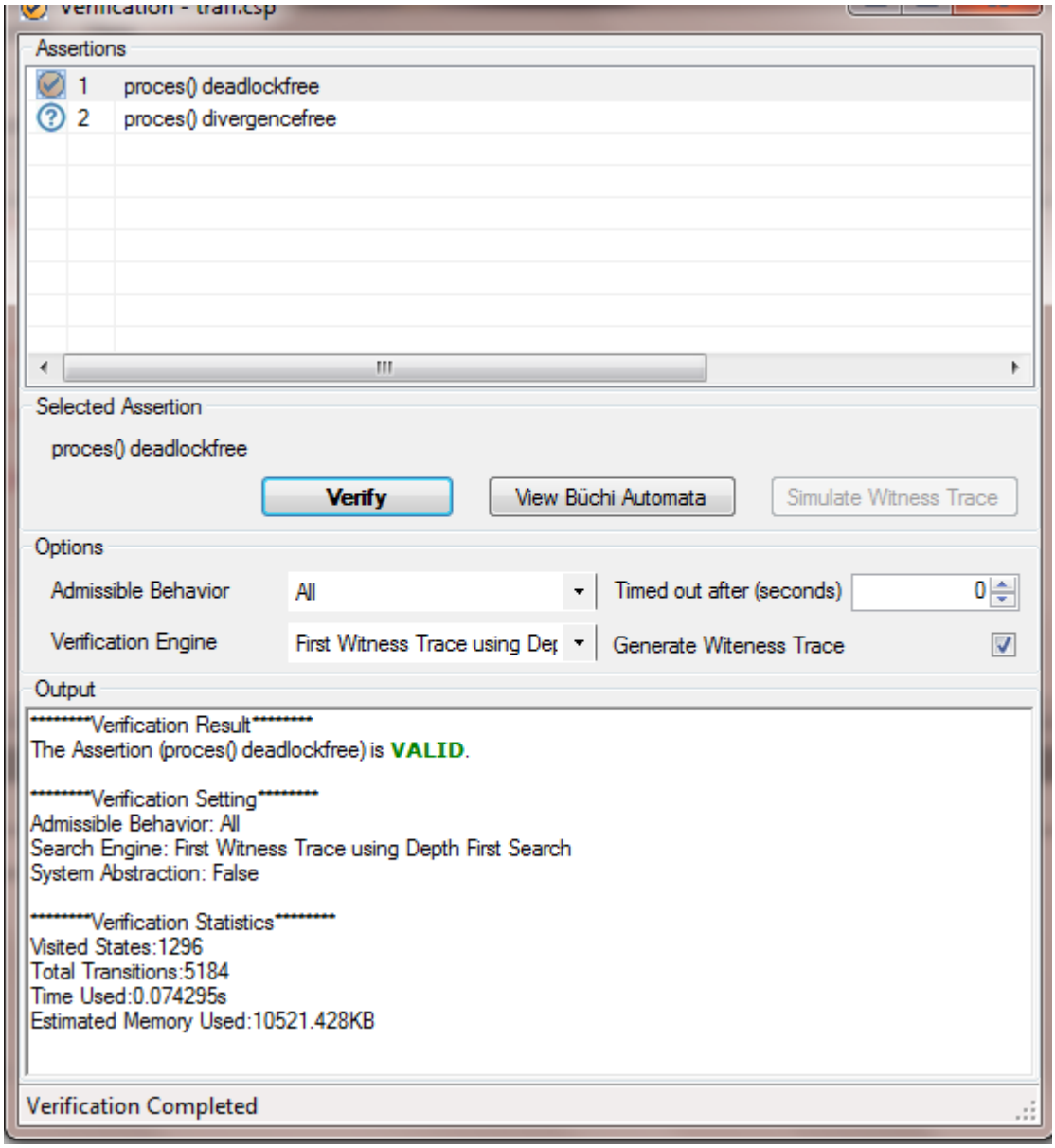


Fig 54: verification of deadlock freeness for the crossroad routes

Deduction and recommendation

From the above simulation tests on the crossroad traffic light system, the coupled network of the roads and platforms have been tested and have been shown t comply with divergence freeness and deadlock. Therefore showing that cars can move at the cross road without two card being on a platform at the same time through the deadlock test.

Chapter 7: Conclusions

DEVS Driven modeling language is a visual language for expressing systems in a simple and expressive way. It has provided an intuitive way to understand systems, specify systems, study and analyze them. Furthermore, with its extensive formal background, DDML provides a means to verify and validate system and check for important dynamic behaviors, properties, structure and operations. The combination of DDML and formal methods in the study of systems provides an analytical, mathematical and logic approach to study possible behaviors of a system and inferring important information about such systems.

The use of formal methods in studying DDML operational semantics has also provided a means to validate the property specifications of DDML against its operational requirement. This provides that opportunity to validate the semantic and operation specifications of DDML along side its graphical model to assure users that the modeling language is sufficient graphically and efficient formally. Different formal methods are mapped to the hierarchical level of abstraction of DDML in order to capture the different semantic and operational information important at each level.

Furthermore, tools applicable to the formal methods are consequently used to assess the DDML framework at each level using relevant example case study. This provides the means to formally analyze the modeling language. The formal analysis of the modeling language shows DDML provides important properties that are validated through the use of formal tools. Properties such as safety, progress, eventuality, reachability, completeness, precedence and lack of deadlock have been shown through DDML case studies showing that DDML is robust and versatile to provide such properties to users.

The major challenge of this work is the need to understand each formal method and formal tool in order to use them in accordance to DDML. The study of the mathematical framework of each formal method is required and the need to be versatile with the writing of specification using these methods is essential. It is also important to study each formal tool with relevant literature examples in order to learn how to use the tools, be conversant with the tools so that it would be easy to align the use of such tools to DMML relevant case studies.

It is pertinent to know that this approach through the use of formal methods and relevant formal tools have provided a strong background for formal analysis of DDML. In addition, it proffers a means to study DDML in the future by verifying models of systems to achieve verification, validation and accreditation of software and hardware systems. This can be achieved by integrating the different tools used at each level in order to provide a single tool that can easily analyze systems formally and validate their properties. Integrating the different formal tools provide a single software tool that can easily and automatically analyze any DDML case study on the run and provide significant information about its properties, structure and behavior at all levels of abstraction.

In the future, we hope to use the formal translation of DDML semantics to the formal tools to help develop a tool that can easily translate DDML formally to different formal method as well ass assess the different properties available at each level of abstraction. This tool should automatically generate codes from DDML to the different formal methods such as LTS/FSP, Kripke Structure, LTL, CTL and CSP. And

with the aid of the architecture and structure of open source formal tools used, be able to create an integrated tool that can asses and validate properties at all levels. Below is a structural diagram showing the architecture of the proposed DDML formal translation and analysis tool.

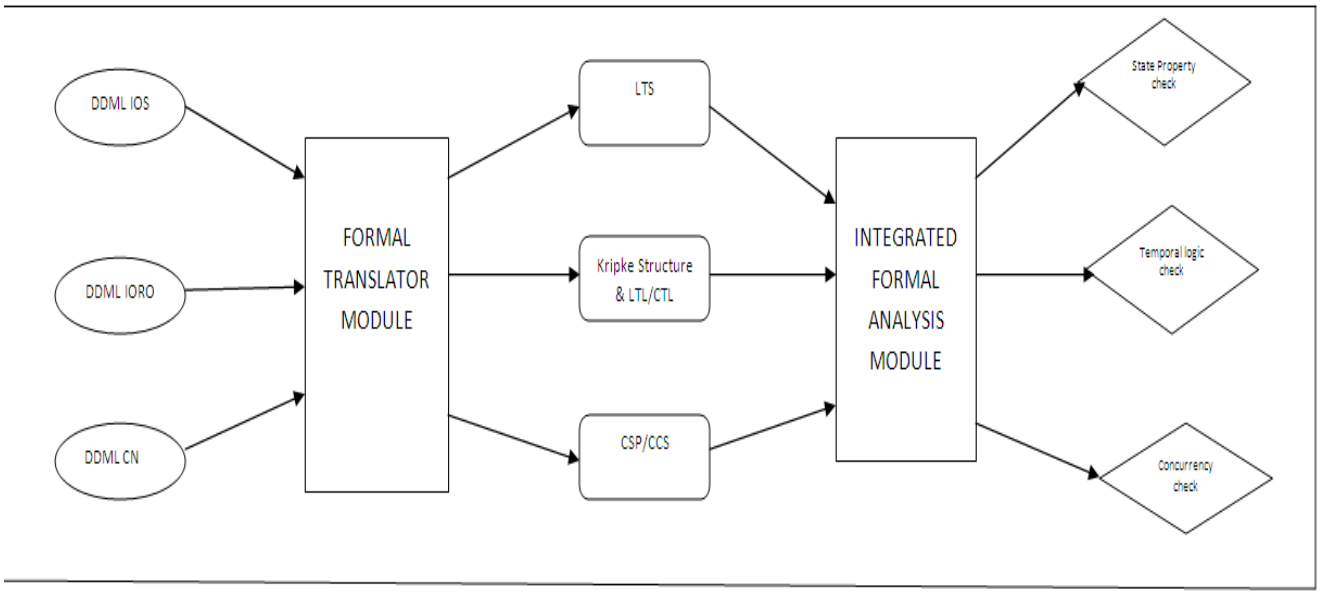


Fig 44: The proposed DDML formal translation and analysis tool architecture

Reference

1. Literature review of air traffic controller modeling for traffic simulations De Prins, J.; Ledesma, R.G.; Mulder, M.; van Paassen, M.M.; Boeing Res. & Technol. Eur., Madrid Digital Avionics Systems Conference, 2008. DASC 2008. IEEE/AIAA 27th 26-30 Oct. 2008 St. Paul, MN
2. European Transport Conference 2003 Innovative Methods in Transport Analysis, Planning and Appraisal > Experimental Economic Models. Distributed behavioral simulation: an innovative way to carry out traffic studies. A Champion, CS-SRILOG; J-M Auberlet, S EspiŽ, INRETS, FR
3. Combining DEVS and Logic. Mamadou K. Traoré. LIMOS CNRS UMR 6158. Université Blaise Pascal. Campus des Cézeaux
4. [Stevenson 2003] Stevenson, D.E. From DEVS to Formal Methods: a Categorical Approach. *Proceedings of SCSC'03, Summer Simulation Multiconference*, Montreal, Canada: 1-6. 2003
5. [Kuhn et al. 2003] Kuhn, D.R., D. Craigen, and M. Saaltink. Practical Application of Formal Methods in Modeling and Simulation. *Proceedings of SCSC'03 (invited). Summer Simulation Multiconference*, Montreal, Canada, 2003.
6. Zeigler, B., Praehofer, H., Kim, T. 2000. *Theory of Modeling and Simulation*. 2nd Edition. Academic Press.
7. FORMAL FRAMEWORK FOR THE DEVS-DRIVEN MODELING LANGUAGE. Ufuoma Bright Ighoroje , Oumar Maïga, Mamadou Kaba Traoré.
8. Semantic Mapping of DDML Coupled Networks to CSP. . Ufuoma Bright Ighoroje , Oumar Maïga, Mamadou Kaba Traoré. July 31st, 2011
9. Process Analysis Toolkit (PAT) 3.4 User Manual. Liu Yan, Jun Sun, Jin Song Dong. July 2007.