Temporal Model Verification and an Extension to System Identification

Mehmet F. Hocaoglu* Cuneyd Firat* Bernard P. Zeigler** Hessam S. Sarjoughian***

* Information Technologies Research Institute, Tubitak Marmara Research Center, Gebze, Kocaeli, 41470 TURKIYE, {hocaoglu, cuneydf}@btac.mam.gov.tr

** Arizona Center for Integrative Modeling & Simulation (ACIMS), 1230 E. Speedway Blvd, Tucson, AZ 85721-0104 / USA, zeigler@ece.arizona.edu

*** Computer Science and Engineering Dept. P.O.Box 875406 Arizona State University Tempe, AZ, 85287-5406 Hassam_Sarjoughian@asu.edu

Abstract

In this paper, temporal logic (TL) based simulation model verification and an extension to a novel system identification (SI) approach and agent learning are presented. Both model verification and system identification accommodate Discrete Event System Specification (DEVS) and combine DEVS with the ability of TL to capture important behavioral specifications for models. Whereas, in model verification, temporal statements stated in design phase are used, in SI phase, instead of using them, the software developed in the study constructs temporal expressions that state behavioral characteristics of the system by collecting execution results. Input of the tool is a simulation model or a model exhibiting time sequenced behaviors that is unknown internally and accepted as a black box.

The paper summarizes three central concerns of the tool, as a verification tool, a system identifier and proposes a research direction to learning ability for intelligent agents. In verification concern, the tool checks whether or not the model being executed satisfies the design requirements that are represented in temporal formulas form. In system identification, temporal formulations of the model being executed are achieved. The identifier looks for solutions how to construct the simulation model based on temporal statements looking at the execution results, their finite state counterparts and eliminates ambiguous expressions. This capability also allows modelers to compare the model they desired with the one the tool constructed. The fact that an identified system by temporal formulas ensures deeper understanding is an advantage presented by system identification purpose. The idea is expanded through the behavioral systems such as agents and it establishes a base for agents sharing the same environment to learn behavioral patterns of each other. In verification process, the approach allows modelers to compare the model, if they designed, with the model or models.

Keywords: Logic Programming, Reverse Engineering, System Identification, Temporal Reasoning, VV&A

1. Introduction

The main aims of the study are to introduce temporal logic based model verification and extend it to a novel system identification approach temporal logic (TL) by combining the expressive power of DEVS with the ability of TL and the ability of logic programming paradigm. While combining DEVS with TL undertakes the task capturing important behavioral specifications for models, logic programming paradigm makes parallel and symbolic thinking possible to allow model construction by observing model behaviors. Since an identified system assures a modeler to compare his/her model that is supposed to behave the modeler expect and satisfy the requirements declared in temporal formulas form with the model that is actually executed and constructed by behaviors got from the model execution, system identification purpose also keeps a verification purpose.

The purpose of system identification is to make an unknown system known by formulating its model based on temporal formulas by tracing its behaviors. The approach tries to guess what the model that generates such
behaviors is behind of the system and formulate them in some temporal formulas patterns. Since identification of a system also means decoding its behavioral patterns in any type of automaton representation, modelers or any other software sharing any common environment have ability to learn the decoded system behavioral patterns. The learning ability is more important for agent software because they watch and affect environment, and generate contrary behaviors against the behaviors generated by other agents. The capability that is known as learning gains more importance in the case many agents share a simulation environment to succeed some goals [1]. The approach makes facing to modelers via system identification and verification and software via learning.

The purpose of verification is to assure that the conceptual model is reflected accurately in the computer code. In other words, verification is the act of proving or checking that a formal system has a formally stated property. In the strictest interpretation of the word, the goal is just to find rigorous evidence that the system is correct in the sense of having this property [2]. It quite often involves some degree of abstraction about system operations, and/or some amount of simplification of actual operations [3].

A typical framework for computer-aided analysis or verification has the following four components [2]:
- A formalism for modeling the system,
- A formalism for stating analysis questions or properties,
- A formal meaning for the relation “the system has the properties”,
- An algorithm for checking whether a given system satisfies a given specification.

For the simulation modeling, the effort spent to prove the identity between modeling requirements that modelers want the system to do and behaviors that the system exhibits is known as simulation verification process. When the four components for the framework construction are considered, our verification framework in this study employs DEVs [6] as modeling formalism, Temporal Logic as property specification formalism, it satisfies relation of temporal logic as “the system has the property” relation and pattern checking algorithm implemented in logic programming as the algorithm for checking whether the behavior matches with the stated properties. Accommodating temporal logic in software verification area is offered by Manna and Pnueli [4] and in simulation domain by Hong and Kim [5]. Instead of the classical direction of simulation verification from property statement, modeling, execution and checking, the approach just takes the resultant state space into consideration and generates simulation model design by formulating the behavioral system specifications in temporal formulas (TFs) form. On the contrary of traditional approaches, the proposed approach constructs a new model or models from execution result without taking the actual model (being executed) and its requirements into consideration. In this case, the verification is done by comparing the model being executed with automatically constructed one(s). The tool is a further step from the earlier study named “Temporal Verification Framework” [6], because it is capable of not only testing consistency of a given requirement in TF form with execution trajectory but also generating a model or a series of models meeting the requirements. Generation of models for the system being executed is related with system identification, which is a capability surrounding verification process, making internally unknown system known, namely, decoded.

In engineering domain, the word “reverse” means constructing an engineering artifact or a model from its output or behaviors they present. In fact, a strict relation is seen between system identification and reverse engineering. Both in identifying a system and a reverse engineering activity, modelers deal with encoding the system via its outputs recognized by back-end interfaces. The system generates behaviors representing its internal characteristics keeping temporal, logical relations being able to be formulated between state variables that are spread along with time axis. The purpose of system identification claims all these relations can be formulated in TFs form and discovered by an inference engine which has a rule base constructed by generic TF formulas patterns and a search mechanism working on state space.

In software engineering domain, reverse engineering concept means getting software design from source codes and documenting it in any format. Because simulation models are put forward as software products, the same interpretation is acceptable for simulation domain. The idea of reaching the source model from its products or results is the notion, also, accommodated in this study by aiming to construct a new model or models as a counterpart of the model currently being executed via inferred relationships among state variables in resultant state space. That is the reason why the process followed by Temp-SI can also be called as reverse engineering process.

The root of accommodating temporal reasoning in system identification and
verification goes over to software verification by temporal reasoning and to simulation model verification from there. Temporal logic based model verification has been used by many researchers for the diversity of verification problems in real-time or non-real time systems [7],[8],[9],[10]. We see that temporal logic based verification is applied for verification of asynchronous circuits [11], verification of reactive systems [12] and real-time program synchronization and verification [13]. In literature, temporal logic is used to extend modeling tools such as Petri-nets and finite state machines for functional testing [14],[15].

The paper is organized as follows. Section 2 is reserved for languages and automata theory because of its strict relation with system identification. Since simulation models are built based on Discrete DEVS formalism and simulation model designed by TFs, Section 3 is reserved for DEVS formalism. Section 4 is about basics of logic, temporal logic and logic programming. The proposed approach is explained with ideas behind the algorithm, the algorithm, architectural view and search strategy in Section 5. The approach is tested by some examples and the experimental results are given in Section 6 and the results are discussed in Section 7. The last section gives a research direction to follow this study up.

2. Languages and Automata Theory

The theories of languages and automata are formal ways to study the logical behavior of Discrete Event System (DES). The starting point is that any DES has an underlying event set E associated with it. The set E is thought of as the alphabet of a language, and event sequences are thought of as “words” in that language.

A language L, defined over an alphabet E, is a set of strings formed from events in E. A language may be thought of as a formal way to describe the behavior of a DES. It attempts to specify all admissibility conditions over sequences of events the DES is capable of processing, while bypassing the need for any additional structure. The validity of a word is determined by the language’s grammar rules and whether the events that constitute the word are members of the event set E. Grammar rules are patterns determining how event sequences are formed and event sequence repetition rules and also they assure even though any length of the word ends at a final state. A final state is not only a state that the last event of the word created to succeed a specific task carries the system, but also it is a stable state reached after a series of events constructing a word implemented. Final states are not declared directly by modelers, they are implicitly expressed during system definition that define how the system to work. In the verification framework, grammar rules are obtained by decomposition of temporal formulas in terms of temporal operator decomposition rules shown at lower level in Figure 5. Grammar rules are also known as behavioral rules or Production Rules (PRs) as in simulation literature because they represent the system behavioral pattern.

An automaton is a device that generates a language by manipulating the alphabets (events) according to grammar defined as a set of rules. Finite-state automaton (FSA) is an automaton with a finite event set E (alphabet), a state set X which has a distinguished initial state (Xo), a transition function (f) to ensure transition among states depending on the events and obeying grammar rules and a set of final state (F). To visualize, FSAs are shown in graphical through a State Transition Diagram (STD). A STD is constructed by circles and arcs denoting states and event among states, respectively. An arc between two circles represents a state transition between the states that the circles denote.

Premsises of system behavioral specifications consisting of value assigned state variables and declaring transitions among states (an example shown in Premise 1) imply how the system behaves. Ignoring events letting state transitions among states in a premise alleviate modeling workload by making modelers focus on states. A valid word generated by the system must satisfy related grammar rules and each grammar rule is depicted by an FSA counterpart. It can be seen grammar rules ends at a state marked as final state by filled circles in Figure 1, Figure 2 and Figure 3 after completing a series of events and a state transition by each event.

3. DEVS Formalism

DEVS formalism was introduced by Zeigler is a systems theoretic approach to describing discrete event systems [16]. In DEVS formalism, it is necessary to define basic models and how these models are connected. DEVS atomic model represents a basic component’s dynamic behavior and has a well-defined mathematical and modular structure [17], models expressed in this formalism effectively express the operational perspectives of real systems and they are able to represent both structural and behavioral features of systems. DEVS also supports creating complex models called coupled models by coupling several atomic models.
DEVs can be seen not just as a simulation formalism, but also a strong candidate for a more general computational formalism to support system design. To do so, it would have to support, or allow embedding of, other formalisms that are more tuned to analysis and design. System design involves modeling, simulation and analysis of candidate systems. The framework employs discrete-event worldview to systems modeling in which logical analysis, performance evaluation, and implementation can be performed— all within the DEVs formalism. The core of framework that is DEVs formalism is surrounded by Object oriented simulation, Real-Time model execution and model acceptance checking as a first layer shell. The components enumerated above have direct interaction with the core. The shell is also surrounded by another layer consisting of Logical analysis that interacts with model acceptance checking, performance analysis that interacts object-oriented simulation, and implementation that interacts Real-Time model execution. The components also interact each other in enumeration order in a fashion that defines a circle. The outer layer surrounds these layers as an implementation layer [16].

In addition to all the properties enumerated above, DEVs allows developers to define states and state transitions. Each atomic model may have several state definitions and transitions between states are ensured by allowing sending an event to an atomic model in any state to shift it into another state. Since Finite State Automatons (FSAs) are concerned with discrete states and the events allowing the model to navigate between states, the representational similarity between FSA and DEVs implies the incorporation of DEVs based model analysis and verification using language theory in this study. Simulation models developed based on DEVs formalism generate timed state information and the job of system identifier to do here is to discover TFs among these state variables taking time stamps and the values into consideration and matching languages grammar rules. This capability of DEVs allows modelers to construct models directly from finite state automatons.

4. Logic Programming and Temporal Logic

A central concern of logic is to take a situation described by a particular set of statements that are assumed, supposed, or otherwise accepted as true and then to determine what other statements must also be true in that situation. These other true statements are implicit in that situation and are, thus, said to be implied by the original ones. Thus, logic can be used to make implicitly true statements explicitly. The original statements are called premises, the “new” statements are called conclusions, and the process of making conclusions explicit is called inference [18]. The inferring unknown depending on known—a set of premises, a conclusion inferred from them and the rules for determining when a statement is true in a situation are among the concern of semantics—known as deductive logic.

Programming paradigms have their own worldviews that they use to represent the world. Object oriented paradigm is based on accepting the world constituted by objects that have attributes that makes them distinguished from other objects and behaviors that they implement to fulfill some tasks. Functional programming focuses on functions instead of the actors that implement them. According to the functional worldview, the world is constituted series of functions. The relations among objects rather than their themselves and functions, a proof theory and facts of related world are the fundamental components of Logic programming (LP) paradigm. LP can be broadly understood as the use of logic to represent problems and problem-solving methods, together with the use of appropriate proof procedures for the effective solution of those problems. For the most part, logic programming today uses Horn-clause logic augmented with negation-as-failure to represent knowledge and employ backward reasoning to solve problems by problem reduction [18]. Problem reduction is to convert the problem into simpler and simpler one until reaching a base that is the most simple and has a known solution. LP is a simple yet powerful formalism suitable for computing and for knowledge representation. Its root in automated theorem proving, from which it borrowed the notion of a deduction [19]. The paradigm is summarized by the following three features; (1) values are assigned to variables by means of automatically generated substitutions, called most general unifiers, (2) the control is provided by a single mechanism: automatic backtracking and (3) computing takes place over the domain of all terms over some alphabet.

Predicate logic is based on an ontological commitment to the existence of objects and relations in the world, and even, in simulation models. While components joint to simulation models are object models, state variables are the facts of the world and they constitute the simulation trajectory. Since languages that state how the system behaves keep relations among state variables, their representations are made in predicate form. Envisioning the world as objects and relations both logical and behavioral guides us to choice object-oriented for simulation and LP for logical operations programming.
paradigms. As mentioned before, “the ontological commitments” here that point out the behavioral relations between objects and relations with rest of the world are formulated in TFs form and processed as predicates. Because of the object-oriented technology surrounding DEVS it is a good choice for simulation and so is Prolog for logical characteristics enumerated above matching our expectations.

TL extends ordinary predicate logic by introducing a set of special temporal operators that provide a natural, succinct, and abstract description of precedence, invariance, and frequent recurrence of events in time. In this sense, TL is a part of modal logic. A modal is an expression (like ‘always’ or ‘eventually’) that is used to qualify the truth of a judgment. Modal logic is, strictly speaking, the study of the deductive behavior of the expressions ‘it always … that’ and ‘it eventually …’. Narrowly construed, modal logic studies reasoning that involves the use of the expressions “always” and “eventually” [20]. With Temporal Logic one can specify how components, protocols, objects, modules, procedures and functions behave as time progresses. The specification is done with (Temporal) logic statements that make assertions about properties and relationships in the past, present, and the future. In addition to that, TL provides operators for reasoning about program computation. A computation is a sequence of states that can arise during program execution. Informally, the first state in a computation represent the present, starting at the beginning of the program, so a future state in one computation can be the present state in another [21].

The fact that a first-order language over interpreted symbols for expressing functions and relations over some concrete domains such as integers, arrays, and lists of integers is an underlying assertion of language L. We refer to a formula, which is selected as TF in this study, in the assertion language L as a state formula, or simply as an assertion [22]. For an assertion p and a state sequence s that interprets all variables appearing in p, we write \( s \models p \)

To denote that s satisfies p, which can also be described by saying that p hold on s, or that s is a p-state.

5. The Idea Behind The Approach

As stated earlier, simulation models executions generate state trajectories and each trajectory is constituted by time and value labeled state variables. The trajectory defines a Close-World that state variables are accepted as facts of that world. The facts are represented as StateVariableName(Value, Time) and the rules constituting the language, in other words, relation among the world objects as RelationName(ParametersList):-Constraints.

Constraints are a set of facts and relations connected each other by logical and temporal operators. All conclusions in this world are inferred from state space and in the guidance of decomposition rules of temporal operators and TF rule pattern (TFRP) premise rules.

To liven the idea up, a software tool is developed named as Temporal System Identifier (Temp-SI). The ideas leading the study of developing the tool are summarized under the seven points. These are;

1: Time is a problem domain independent common concept that brings temporal reasoning and simulation together on the same platform.

To be able to develop a problem domain independent temporal verification tool, common concepts must be found to meet temporal reasoning and simulation with each other. One of the concepts must be time because it is ripe for the domain-independent (generic) variety to be developed for application to the large-scale complex modeling [23] and, from temporal reasoning point of view, it is a representation base for frequency of event recurrence and precedence. Both temporal reasoning and simulation have time representations, although their representation fashions differentiate. When the representations are combined, models that have richer time representation both qualitative that allow giving precedence among the events and state transitions as temporal formulations express and quantitative that allow defining process durations and timely event handling as is done in simulation modeling. State variables used in temporal and simulation modeling are also another common point that defines a shared modeling platform.

2: Verbal system behavioral specifications and temporal premises are transitive. Verbal system behavioral specifications can be translated into temporal premises and vice versa.

Temporal formulas and the operators are based on human natural languages. The formulas are derived from verbal system behavioral specifications. In system design, behavioral specification can be seen as requirements because they state what the modeler/customer expects the model to do. When constructing a simulation model, system behavioral specifications derived from requirements stating how the model works (behaves). The system behavioral specifications
are strictly related with deducing state transitions the system to be modeled exhibits. The statements take target arranging the behaviors (the system progress) on time axis in ordered fashion without giving any exact time constraints. For example, a system behavioral specification premise might be such as

Premise: “If the Assign Weapon System is in Stand by mode, it will eventually change its state to Active mode”

The formulation of the premise is given in Formula 1 (F1) and the conversion steps transforming the premise into its temporal formula counterpart are summarized below.

Assign Weapon System Status (Aws) is a state variable taking values from a limited state set such as /Stand by, Active, Passive, Broken/. The premise is symbolized as:

\[ \text{Aws}= \text{Stand by} \rightarrow \text{Aws}= \text{Active} \]

We know that state transition is a time consuming task. Even though we do not know how much time necessary to change the state but we are sure it is necessary.

\[ \text{Aws}= \text{Stand by} \rightarrow \text{eventually (Aws}= \text{Active}) \]

Other information is that this state transition is repeated though simulation execution, namely, it is done always.

\[ \square (\text{Aws}= \text{Stand by} \rightarrow \Diamond \text{Aws}= \text{Active}) \quad (F1) \]

Knowing the Assign weapon system will change its state is just being aware of the world truth but giving a qualitative time span is to arrange the progress on time axis. Even though, the expression does not give any exact time about how the change will happen, but its qualitative interpretation gives a progress order by eventually logical modal. Expressing the progress in qualitative way is very important property because no modeler can know time constraint at the very beginning of the system design and even exact time information is related with validation not with verification.

Bridging the gap between requirements expressed in natural language statements and TFs and between TFs and simulation modeling temporal operators interpreted in natural language. The effort makes the concept clear from the point of suitability of temporal reasoning to simulation concepts. Although temporal operators are grouped past and future operators, focusing on how the system being modeled to behave in simulation model development is a forcing factor to focus on future operators.

“Eventually” operator allows modelers to express there is a time span between the event following each others, in parallel state transition, and represented in TFs on the left and right sides. The event represented in right hand side happens some time later the event at the left hand side happens. Always operator expresses a repetition on the event it is applied. If it is applied to a behavioral specification, as seen in Formula 1, always operator ensures the transitions that the specification states happens along with whole simulation execution. The “Next” operator clearly states values of state variables that the operator is applied at next time step.

The operator “not (¬)” has a designative role on FSAs. In a way that each component, which simulation model has, has a limited state set or unlimited sets that is divided into meaningful limited sets and if a premise stating that a state variable will not be equal to a specific value some time later, the premise imply that the variable will have a value from the rest of the state set. Being not equal to a specific value carries on a meaning being able to equal to rest of the state values in the state set. This is the separating the world into two isolated parts; the first part is being equal to something, and the second one is rest of the world (not equal). The important point is that it is not necessary whether or not the TF has a “not” operator because even it does not have, production rules have. That is the reason why FSA counterparts of TFs have just two state definition abbreviated as S0 and S1. S0 and S1 keep state variables and, although, state variables have different values from state transition to transition, all values state variables have to satisfy the TFs.

3: The behavioral specifications that are translated into TFs can be decomposed down to grammar rules, every grammar rule is represented by a FSA and a FSA ends with a finale state.

Decomposition process is to decode temporal premises into simpler logical premises such as “and”, “or” by using temporal operator decomposition rules (seen in Table 1). The newly generated premises are known as production rules because they govern production of valid behavioral patterns, which are named as words in language and automata theory, of models. The words should comply with FSA representation of grammar rules ending at final states. Decomposition of temporal rule patterns and their FSA representation are shown below for some rules. The filled circles in Figure 1, Figure 2 and
Figure 3 represent final states (S and P stand for initial state and production rules in the decomposition processes, respectively). The production (grammar) rule patterns created as a result of decompositions of TFs are summarized in Table 2.

4: The main capabilities of LP most general unifiers and automatic backtracking (Section 4) make defining temporal formula patterns (families) possible.

Temporal formula patterns are formula families that keep many temporal formulas constructed using state variables in a simulation trajectory. To make state variables of the simulation trajectory matched with the pattern variables a search process is started up via automatic backtracking and matching state variables are assigned to the variables via general unifiers. This property allows very high, parametric level solution definition such as finding a state set related to a set of state variables so that they satisfy conditions represented as A at time T0 and B some time later at time T1 (A→◊B). This not only makes solution algorithms close to verbal definitions, but it covers many temporal formulas stating how system works more than modelers can think.

TFRP #1: ϕ:= □(A→◊B)
Step 1: A→◊B, S ←ϕ
Step 2: A→◊B = ¬A+ A(◊B) (D7-Table 1)
Step 3: P:={S→A, S→A(◊B) }
Step 4: ◊B→B + ¬B(◊B) (D2-Table 1)
Step 5: P:= P∪{◊B→B, ◊B→B(◊B)}

Figure 1: FSA Representation of □(A→◊B) pattern

TFRP #2: ϕ:= □(A→□B)
Step 1: A→□B, S ←ϕ
Step 2: A→□B = ¬A+ A(□B)
Step 3: P:={S→A, S→A(□B) }
Step 4: ¬□B→¬¬B(¬□B)
Step 5: P:= P∪{¬□B→¬¬B(¬□B)}

Figure 2: FSA Representation of □(A→□B) pattern

TFRP #3: ϕ:= □(A→□B)
Step 1: A→□B, S ←ϕ
Step 2: A→□B = ¬A+ A(□B)
Step 3: P:={S→A, S→A(□B) }
Step 4: ◊B→B(◊B)
Step 5: P:= P∪{◊B→B(◊B)}

Figure 3: FSA Representation of □(A→□B) pattern

5: Temporal relations are kept among state variables in a simulation trajectory. The relations form temporal formulas reaching up to system behavioral specifications by using time and value labeled state variables in simulation execution trajectory.

The fact that a state trajectory keeps some relations among its state variables being able to be expressed in verbal and to be transformed into TFs is another idea behind the process. The implementation of the idea is to find out the relations and express them in temporal formulas. Temp-SI accepts state variables and forms temporal formulas stating system behavioral specifications by a series of inference engines (IEs) that handle decomposition rule base that keeps decomposition rules of temporal operators, production rule patterns rule base that keep production rule patterns obtained from decomposition of temporal formulas and the rule base that keep temporal formula patterns. Since the production rule base is created by decomposed TF rule patterns using decomposition rule base, it is a dynamic rule base, which is created in run time. The expression “pattern” here expresses that the rules are represented in the rule bases are not initialized by state variables, namely, they do not have any state variables. They are highly abstracted and the variables they have are initialized in run time by state variables as a result of a search process done on state trajectory by collecting them on a search tree. The variable A and B shown in Table 1 are initialized by single or combined state variables and state variables are initialized by their value sets. The expression “Combined state variables” is used to express the state variables connected each other by logical operators.

6: Logic programming paradigm allows us to model temporal premises directly from their natural language interpretations.
Logic programming paradigm aims to develop codes how to think instead of coding a solution procedure step by step. This property makes code development task close to verbal problem definition rather than following stepped solution procedures. Parallel thinking and symbolic computation properties that the paradigm promises are the constraints to be satisfied. Managing several rules, state variables in parallel and symbolic value state variable take, and temporal and logical operators that are in symbolic nature are the factors that strengthen usage of logic programming. One of the most famous logic programming language Prolog is used to develop the tool.

7: Reversible rules and program fragments.

Parallel logic and symbolic thinking brought by logic programming allow to create production rules from behavioral specification and search simulation trajectory to find words satisfying the rules for given state variables or to find state variables constituting words that satisfy production rules that are collected as temporal formulas. That makes reversible rules, namely, that their inputs and outputs are interchangeable. The idea leads inferring temporal formulas by discovering state variables with their values so that they satisfy production rule patterns and temporal formula patterns.

5.1. The Tool Architecture

The tool is constituted by different layers that are related to serve each other. The layers consist of Rule bases separated into three parts in different abstraction levels, Inference engines handling them, a front-end interface picking execution state variables connected with simulation model and a back-end interface keeping and showing the results. The rule bases placed at bottom, middle and top layers are named as “Decomposition Rule Base”, “Production Rule Base” and “Temporal Formula Patterns” and shown in Table 1, Table 2 and Table 3, respectively.

Table 1: Decomposition Rules

<table>
<thead>
<tr>
<th>Rule #</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>D1</td>
<td>□ A ⇒ A □ A</td>
</tr>
<tr>
<td>D2</td>
<td>□ A ⇒ A + ¬A (□A)</td>
</tr>
<tr>
<td>D3</td>
<td>A □ B ⇒ B + A (A □ B)</td>
</tr>
<tr>
<td>D4</td>
<td>¬A ⇒ A + A (¬A)</td>
</tr>
<tr>
<td>D5</td>
<td>¬□ A ⇒ ¬A (¬□ A)</td>
</tr>
<tr>
<td>D6</td>
<td>¬(A □ B) ⇒ ¬AB + ¬B(¬(A □ B))</td>
</tr>
<tr>
<td>D7</td>
<td>A → B ⇒ ¬A + A B</td>
</tr>
</tbody>
</table>

Table 2: Production Rules

<table>
<thead>
<tr>
<th>Rule #</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>PR.1</td>
<td>S → A</td>
</tr>
<tr>
<td>PR.2</td>
<td>S → A(B)</td>
</tr>
<tr>
<td>PR.3</td>
<td>◻ B → B(◻ B)</td>
</tr>
<tr>
<td>PR.4</td>
<td>◻ B → B</td>
</tr>
<tr>
<td>PR.5</td>
<td>S → A(□ B)</td>
</tr>
<tr>
<td>PR.6</td>
<td>S → A(¬B)</td>
</tr>
<tr>
<td>PR.7</td>
<td>◻ B → B(¬◻ B)</td>
</tr>
<tr>
<td>PR.8</td>
<td>¬B → B(◻ B)</td>
</tr>
<tr>
<td>PR.9</td>
<td>S → A(¬B)</td>
</tr>
</tbody>
</table>

Table 3: TFs Patterns Rule base

<table>
<thead>
<tr>
<th>Upper Layer rule base</th>
</tr>
</thead>
<tbody>
<tr>
<td>(A → B) := PR.1 ∧ PR.2 ∧ PR.3 ∧ PR.4</td>
</tr>
<tr>
<td>(A → B) := PR.1 ∧ PR.6 ∧ PR.7</td>
</tr>
<tr>
<td>(A → ¬B) := PR.1 ∧ PR.5 ∧ PR.8</td>
</tr>
<tr>
<td>(A → ¬B) := PR.1 ∧ PR.8 ∧ PR.9</td>
</tr>
</tbody>
</table>

As mentioned earlier, the effort exhibited by the tool is to discover the temporal relations being kept among state variables. The task is fulfilled by layered three rule bases and a series of inference engines. The bottom layer consists of the rules related with how fundamental temporal operators are reduced into logical premises expressed by more basic ones such as by “and”, “or” and “not”. The inference engine of that layer is in charge of collecting state variables together with their all-possible values from resultant execution state space and arrange them on search trees. “Or” and “And” that are the fundamental logical operators forming the search trees shape a bifurcation point and a sequential path, respectively. All state variables are collected on a search tree in “or” form and their values follow the related state variables in “and” form but the values of a state variable are arranged in “or” form according each other. Time is another important constraint in constructing a search tree. Wherever a temporal operator takes part, search trees collect future values for those variables for the variable B.

The responsibilities among rule bases are shared so that Decomposition rule base at the lower layer, Production Rule base (Grammar Rule base) at the middle layer and TFs Patterns at the top layer are in charge of (1) Decomposing a temporal premise into logical premises and (2) collecting state variables and their values on search trees, testing whether a state variables set forms a valid word, namely, exhibits a valid behavior and forming search trees by lower layer search trees satisfying production rules or collecting state variables in a way that they form production rules, (3) collecting all satisfied production rules in a way
that form a temporal formula fitting one of TFs patterns, respectively.

The rules at all layers have two ways usage. They can be used not only to test whether or not a state set satisfies the rules but also to find the state variables set that satisfies the rules. For convenience, the usages for testing a given state variable set with their values and for searching state variables constituting a temporal formula or a production rule patterns on execution resultant state set are named as “Bottom-Up” and “Top-Down”, respectively. Whereas Bottom-Up usage is the approach followed in “Temporal Verification Framework”, Top-Down is a base for System Identification. The initialized state variables by temporal system behavioral specification premises are characteristic of Bottom-Up approach. What state variables, for what values to be searched in a way that a given PR to be satisfied is known at the beginning of the tool execution. In Top-Down, in other words, System Identification, state variables, their values and even TF patterns themselves are unknown and they are parts of search process.

Each rule base has its own inference engine. Those engines are independent from how many rules the rule base has and because they decompose the rule patterns from bottom to top and top to bottom, it has a possibility making the tool flexible to populate the rule bases by adding new temporal operators. The procedure is as following:

- Add decomposition rule of the new temporal operator to decomposition rule base,

- Add temporal formula rule patterns to temporal formula rule knowledge base consisting of the new temporal operator. The production rule base is created in run time by its inference engine using decomposition rule based on TF pattern.

The different search directions make interfaces to the same component changed from front to back or vice versa. The interface placed between “TFs set-Model Specifications” component and Rule bases serves as a back-end interface in Top-Down approach, but a parser interfaces with “TFs set-Model Specifications” component by a front-end interface at the same place in Bottom-Up approach (Figure 4). The parser is in charge of parsing TFs so that the lower layer rule bases decompose and test parsed TF premises.

As shown in Figure 4, as a result of layered structure, each layer receives the inputs that it needs from neighbor layers. Decomposition rule base serves as an alphabet for middle layer so does middle layer for upper layer because the production rules are constructed by decomposition rules and they construct temporal formula rule patterns.

5.2. Working Procedure of the Tool

5.2.1. Working with the Tool

The important point, on which an attention should be spent, is that understanding how the tool works is related with comprehending the contents of rule bases, interactions among rule bases, implementation of rules, construction of a solution tree and process inputs, outputs.

State variables that are the only inputs of the process are received from the simulation model being executed via the interface that connects with simulator. In Figure 4, numbered circles starting with letter “T” and “B” show the execution route and named as station, hereafter. T stands for Top-down and so does B for Bottom-up and their implementation steps are explained below.

Top-Down: “Pick all state variables with their values and their time labels by the Front-end Interface (T1), collect them on search trees so that each decomposition rule has a search tree (T2) and collect the solution trees created by decomposition rules on a upper layer solution tree so that the trees are formed by production rules (T3) and create one more upper layer solution trees by TF rule patterns so that the rules constituting the tree are components of temporal rule patterns, in parallel constitute an initial state (T4), create a solution tree for each TFRP. Finally, assert the temporal formulas constructed as solution and display them on the Back-end Interface (T5)”.

Bottom-Up: “Take a temporal formula that has state variables with values via the Front-end interface from temporal formulas database written by the user (B1), determine state variables, their values, temporal and logical operators declared in the TF by parsing it and assert TF as initial state (B2), decompose the TF to create production rules that are initialized by the state variables and related values (B3), check whether the state created in simulation run time variables constitute valid words by using production rules (B4). Check whether or not the production rules are satisfied (B5). If all production rules of the TF are satisfied, evaluate it as the temporal formula and behavioral specification behind it are verified (B6).”
For both usages of the tool, bottom-up and top-down, initial state is a key element for implementation and it is necessary to spend a special intention. Two kinds of initial state definitions are faced. The first one is initial state of simulation execution that is defined by the state variables values at the very beginning, at which execution time is equal to zero, of the execution and it is out of scope for this study. The second one is named as TF initial state that is defined by evaluating the TF as a state. Although, a simulation execution has just one simulation initial state, it has several TF initial states at any time though simulation execution depending on temporal operators the TF has. Since TF initial state can be seen as the first state of a series of time ordered state transitions that the related TF declares, the transition series can be found in anywhere through simulation execution. A transition from the state that satisfy A (left side of the TF) to the state that satisfy B (right side of the TF) is a transition from initial state to any other state. A transition to opposite direction is a transition to initial state. In other words, we know that all TFs are represented by FSAs and FSAs exhibit a series of behaviors starting at an initial state and ending at a final state. Informally, it can be said that the initial state is the state that behavior series start.

In System Identification, state variables and their values are not found at the beginning of the process, the initial state is determined in parallel with constructing the TF by related production rules. Since the thinking way is reverse, the procedure finds state variables, assigns values so that they satisfy production rules. The state variables constructing an initial state are searched in parallel searching following states by taking time gaps between states into consideration. After completing the production rules, they are collected so that constitute TFs.

5.2.2. Search Process and Inference Mechanism

The primary purpose of search process is to create temporal formulas stating behavioral specifications of the system being modeled by making inference. Making Inference is related with checking whether the state variables satisfy production rules that are obtained from decomposition of the TF being verified in Bottom-Up approach and related with constituting TFs by initializing decomposition rules by state variables so that they satisfy production rules that constitute the related TF.

Figure 4 shows the solution tree in parametric structure. While a given production rule obtained from a temporal formula, which has initiated and definite state variables, being verified whether or not the behavior expected the model to perform and dictated by the temporal formula is fulfilled by searching state space that constitutes the leafs of the tree, but in top-down direction, state variables are collected bottom of the search tree so that they constitute production rules. The production rules that are created by decomposing TF patterns and have search trees attached consisting of possible solutions are collected so that they constitute the related TF.

Making a conclusion is to search the possible solutions –from resultant state space- on the search tree constructed by the TFs pattern rule premises on the top layer. Since the depth-first search strategy is applied, the search process starts to search from state variables attached the bottom of the tree for finding the state variables satisfying all the rules that form the tree.

The lower level rules (decomposed TFs) also, in some extend, have an abstracted nature because they are always decomposed according to the content they are filled, namely, what A and B express and how the state variables have been connected each other by logical operators such as “and”, “or”. In this sense, this layer serves to upper layer as a dynamic rule base by creating rules depending on the contents. All rules that the layer has both abstract and decomposed are building block.

We have four TFRPs numbered as 1 through 4. The figure can be expanded starting by top layer so that this expansion results a search tree constituted its leafs by execution
resultant state space. While a bifurcated branching from a premise shows “or” relation between the connected premises, the premises ordered along with a line are connected each other “and” operator. The unfilled arrows represent end of decomposition. Recursive definition on the same operators mean next iteration — points a new value for related state variable labeled further time than currently being had– on simulation trajectory. The recursions seen on TFRPs refer the premises to be satisfied along with simulation execution. This is concluded facing state collections satisfying the same pattern several times in a simulation execution until it ends. In fact, decomposition is not anything else transforming temporal operators such as “eventually”, “always”, “not always” and “not eventually”, into more fundamental logical operators such as “and” and “or”. To make the expansion of the tree shown in Figure 5 clear, Pattern #1 should be expanded by following the lines numbered as 1. The path is collected as

\[ (A \rightarrow \Diamond B) = \neg A + A(\Diamond B) = \neg A + A(B + \neg B) \]

This highly parameterized solution tree has enormous leaves taking part the lowest level of the solution tree by attached resultant state space as alternative solutions. Although, searching process is a time consuming process, LP provides some worthwhile opportunities. LP allows creating code fragments directly from solution description phrases. The opportunity is combined with how the transformation from system definition premises to dynamic rule creation to be done. Each rule has its own program code fragment in one or more logically connected predicates structure.

**Figure 5: Search Trees Structure**

Solution verbal phrase for PR.1 in Bottom-Up approach is as following.

“The state variable Var that is member of state variable set of the temporal formula given might have two values from its values set such as value Y at time T_{initial} that is named as initial state time and X at time T_{time} so that X is different from La (a value assigned to Var in A part of the TF) and T_{time} is greater than T_{initial} While the variable Var never takes the value La for the duration between T_{initial} and T_{time} excluding the limit values T_{initial} and T_{time}, the value La is assigned as initial state value to the variable Var at time T_{initial}”

**An Example for Bottom-Up: Verification**

\[ (\text{Aws=Stand}_{-} \rightarrow \Diamond \text{Aws=Active}) \]

Production Rule:

\[ \Diamond \text{Aws=Active} \rightarrow \neg \text{Aws=Active}(\Diamond \text{Aws=Active}) \]

While Aws has a value active, it takes a value different from active sometimes later and it eventually takes the value active.

In top-down approach phrase is as following:

“Find a state variable Var that might take two values from its values set such as value Y at time T_{initial} that is named as initial state time and X at time T_{time} so that X is a values of its values set and T_{time} is greater than T_{initial}. Search for a value La that the variable Var never takes for the duration between T_{initial} and T_{time} excluding the limit values T_{initial} and T_{time}, the value La is assigned as initial state value to the variable Var at time T_{initial}”

**An Example for Top-Down: SI**

\[ (\text{Var=val}_{-} \rightarrow \Diamond \text{Var=val2}) \]

Production Rule:

\[ \Diamond \text{Var=Val} \rightarrow \neg \text{Var=Val}(\Diamond \text{Var=Val}) \]

While the state variable var has the value val at any time simulation, it takes a value different from val sometimes later and it eventually takes the value val.

The variables in the definition above far from being ordinary variables taking values a limited or unlimited set so that they accept the state variables of model being executed as values both as single state variable with time labeled state value and multivariable in the same fashion connected each others via logical operators.

6. Experimental Results

A simplified Theatre Missile Defense (TMD) example is selected to test the approach offered.
TMD example consists of four components. These are a Threat Generator (TG), Threats (T) generated in execution time, a Prioritizer (P) that is in charge of watching threats and making queries to recognize and assign a priority to react a behavior and an Assign Weapon System (AWS). TG generates threats randomly. P watches and assigns a priority value to the threats. The prioritized threats list is accepted by AWS and choose a weapon and tries to eliminate threats starting from the threat that has highest priority. The state sets of TG, T, P and AWS are \{Active, Passive\}, \{Dead, Threat\}, \{Active, Query, Stand_by\}, \{Stand_by, Query\}, respectively.

Some of TFs constructed by the algorithm from execution results are as follows;

1. $\Box (TG=\text{Passive} \rightarrow \Diamond (T=\text{Threat}))$
2. $\Box (TG=\text{Passive} \rightarrow \Diamond (AWS=\text{Query}))$
3. $\Box (TG=\text{Passive} \rightarrow \Diamond (P=\text{Query}))$
4. $\Box (TG=\text{Passive} \rightarrow \Diamond (P=\text{Stand_by}))$
5. $\Box (AWS=\text{Query} \rightarrow \Diamond (T=\text{Dead}))$
6. $\Box (AWS=\text{Query} \rightarrow \Diamond (AWS=\text{Stand_by}))$
7. $\Box (AWS=\text{Query} \rightarrow \Diamond (P=\text{Stand_by}))$
8. $\Box (AWS=\text{Query} \rightarrow \Diamond (P=\text{Query}))$

TF #3 and TF # 4 are ambiguous TFs and they are combined as

$\Box (TG=\text{Passive} \rightarrow \Diamond (P=\text{Active})).$

The state set has just three values and if the state variable is equal to “Query” and “Stand by” in different times. This is interpreted as being not equal to active because the rest of set has just a state variable. The same process is valid for TF #7 and #8 and they can also be combined as

$\Box (AWS=\text{Query} \rightarrow \Diamond (P=\text{Active})).$

7. **Practical Advantages of the Approach**

The main advantages of the approach are summarized below.

1. **Knowing less principle:** The approach assumes modelers have almost no information about the simulation model and it allows them assuming simulation model as a black box. The only thing they have is simulation execution result generated by the simulation model. All process is proceeded automatically by the tool. This alleviates the users workload of knowing how verification process is done and how the simulation model is modeled.

2. **Being close to native thinking:** Instead of checking message passing between simulation components, and process whether they work properly, the approach aims to describe how model works by temporal formulas easy to translate into verbal premises.

3. **Verification on execution time:** The approach is suitable to verify models both in execution time and after execution using recorded execution trajectory.

4. **Opportunity catching system descriptions missing:** The approach can catch some temporal formulas describing system behavior modelers do not think and declared.

5. **Richer perspective to the model:** Discovering temporal formulas that modelers do not think arise a new opportunity as discovering a novel and re-worded descriptions from different point of view for a known system behaviors declared earlier by the modeler.

6. **Constructing alternative simulation models.** The approach puts at least one, in most cases more than one, simulation model designs. Having alternative model design provides understanding system behavioral characteristics in depth.

7. **In having a model known case, comprising temporal formulas discovered with known ones.**

8. **Concluding Remarks**

The approach undertakes two central tasks. These are system identification by constructing its model, model verification and, as a benefit of them, enhancing understanding the system. The system identification task is to allow modelers to model simulation models from behavioral descriptions. The modelers can get temporal behavioral descriptions of the model they desire to model even from hand-written execution results. That means if modelers write an execution trajectory they desire the model to behave, the tool can generate temporal description of the system capable of exhibiting the behaviors given. Moreover, this gives an opportunity to decode the model that their internal structures are unknown, just, by examining the behaviors they create. In this
respect, the system identification approach is a kind of learning process. In a software environment that is shared by many agent or agent like learning systems, temporal logic based system identification approach can be used by agents to code each others’ behavioral patterns in FSA or any other representation. Learning behaviors of an agent by another agent is not new and named as Endomorphic Systems by Zeigler [24]. This study offers a learning techniques powered by temporal reasoning.

For a model designed, the approach gives verification process support. The support is not only to check whether system behavioral specification in TFs form are satisfied by the execution state trajectory in bottom-up approach but also, in top-down approach, to discover TFs that satisfy the state trajectory and give an opportunity for checking whether or not TFs set generated by the tool are the same with the TFs modelers have constructed in design phase or whether the generated set consists of the set designed.

Probably, the generated set has more TFs than modelers envisioned, because the process aims to clarify the relations among all state variables even modelers have not taken into consideration. This can be seen as constructing total envision for system identification. The completion set ensures different perspectives by discovering correlations among state variables that although the modelers do not plan at the design phase and this provides modeler with deeper understanding of the system.

The gap between requirements that are declared in verbal and the model designed is bridged by interpretation of temporal operators in natural language and a solution approach is developed close to human thinking by algorithms and coding in logic programming paradigm. This makes solution procedures easier than they are. In this context, the study carries an importance also in combining logic and object oriented paradigms based solution approaches into one frame.

9. Shortcomings of the Approach

The informative weakness of temporal reasoning is the main shortcoming of the approach. Since precedence between state transitions that temporal reasoning focuses on are not informative enough in the sense of causalities of events and state transition occurrences, all successive state transition can be recognized as a logical relation even there is not any logical relation between transitions. The shortcoming is tried to pass over, to some extend, by introducing a domain knowledgebase (Inconsistency Checking Rulebase shown in Figure 4) keeping information about the domain discovering inconsistent temporal relations. For example, two transitive temporal relations, such as $\text{always}(state1=sa \rightarrow \text{eventually}(state2=sb))$ and $\text{always}(state2=sb \rightarrow \text{eventually}(state1=sa))$, can be detected and they are ignored by the knowledgebase.

Sometimes a state transition may affect any other transition indirectly. The approach may infer some indirect, propagated relations between the first affecting state and the last affected state and construct a temporal formulae. Modelers cannot see these indirect temporal relations easily.

Computation effort is another important difficulty. During simulation execution a huge state trajectory are generated by execution and they are added to temporal system identification solution tree as leaves. Getting the solution tree bigger gives longer computational time.

10. Future Works

The study is open to any improvement. Since temporal reasoning based system identification approach allows decoding system behavioral specifications of systems in TF form, to be capable of doing that in TF format or in any other format is to learn how they behave. Making an agent equipped with that capability allows it learn behaviors of other agents or other objects that exhibit behaviors around it by developing their internal behavioral representation in temporal formulas format transferable to FSAs.

Speeding up the process; as seen in Figure 5 solutions are found by searching a huge tree. Number of state variables, of production rules to be satisfied, of TFs pattern to be discovered and length of simulation execution are the factor making the solution tree width larger. The larger and deeper the tree is, the slower the process proceeds. Instead of depth-first search strategy that the algorithm developed in the study uses, faster search strategy such as A* search strategy might be used to make the process speed up.

The premises taken part on the two sides of a TF should be improved to be capable of the representing complicated premises that multi state variable are connected each other by several logical operators.

By depicting FSAs of TF discovered on the screen, it can be allowed for visual interaction.

References

[1]. Hocaoglu, M. F., Firat, C., Sarjoughian, H. S., “DEVS/RAP Agent-Based Simulation”, AI, Simulation & Planning in High


