The Tree-like Local Model Update with Domain Constraints

Michael Anthony Kelly

A thesis submitted for the degree of
Doctor of Philosophy at
University of Western Sydney

December 2011

Typeset in Times with TEX and LATEX2e.

Except where otherwise indicated, this thesis is my own original work. I certify that this thesis contains no material that has been submitted previously, in whole or part, for the award of any other academic degree.

M. A. Kelly

December 23, 2011

To my parents

Acknowledgments

Primarily I would like to thank my supervisor, Professor Yan Zhang for his guidance throughout my doctoral candidature, without his goodwill, patient nature and wealth of knowledge I wouldn't have been able to complete this thesis. Further I would like to thank him for his work in securing the scholarship for this research and allowing me to work with autonomy, I am greatly appreciative.

I would also like to thank the members of the Intelligent Systems Laboratory for their support, guidance and advice in directing my thesis work, their input has helped to shape my work greatly for the better. Also I would like to thank the support team at the School of Computing and Mathematics. Their support with securing technical equipment and assistance with administrative tasks has made my work here much easier. I would like to thank the University of Western Sydney for supporting my research with the awarded scholarship and workspace over the last four years. My thanks also go out to my family and especially my parents for their great advice, never ending support and love in a difficult time.

Last but certainly not least I would like to thank all my friends who have kept me sane in an arduous and perplexing time with your good humour, advice and companionship. You're the best.

Abstract

Model update is the logical extension of model checking, allowing automated modification to models found not to satisfy a given property in the checking process [10, 103]. In local model update, counterexamples are derived from model checking sessions where some ACTL formula has been found unsatisfied. By updating the localised models to satisfy the underlying property, we may derive modifications to the original global model. Constraints also play an essential role in describing allowable system behaviour. Variable and action constraints can be defined which describe allowable updates on a counterexample in the update process, extending developer control over what is a valid update on the original system.

In previous attempts, methods of update required the processing of the entire model. With larger scale industrial models, this was not feasible due to the inherent complexity. Further, constraints placed on the model in question were not addressed, and as such critical functionality could be circumvented (*e.g.* breaking a resource deadlock using some method should not cause some critical functionality of the module to cease functioning). In this dissertation, the foundations of ACTL tree-like local model update are thoroughly studied. We define necessary elements of ACTL local model update, describing ordering metrics for determining which updates are simpler with respect to weak bisimulation ordering. Further to this, we look at the link between local model update and belief revision, semantic characterisations for typical updates, analyse the complexity for general cases of update and present the theory underlying constraint automata.

We present algorithms in the form of pseudocode describing earlier formalisations, characterising update cases based on ACTL temporal operators. With this, a stand alone prototype for automatic generation of candidate fixes based on ACTL specifications is developed. NuSMV is used to derive counterexamples which can be parsed and analysed for generating candidate fixes. To test the effectiveness of the prototype, we present three case studies including a case study containing constraint automata compliance in the SPIN language.

Contents

List of Algorithms

List of Figures

Chapter 1

Introduction

1.1 Introduction and Motivation

As most of us experience, computing has applications to almost every element of life in the modern world. We rely on many of these applications to deliver a high standard of quality, and in the case of critical systems where human life may be involved, be fault free. Saying this, we find ourselves in a position with the hardware and software systems that support computing, where complexity is the status-quo instead of the exception [3]. It is very easy for human error to occur in the design process and errors can be very difficult to detect in the design. Developers require advanced means of computer aided modification to guarantee accuracy and design soundness when designing systems of industrial scope.

Many approaches to computer aided modification and diagnosis exist and have been researched extensively over the last thirty years. Some such techniques are theorem proving, model checking, model based diagnosis, automated repair and model update. Of these, arguably the most promising approach has been model checking with its automatic approach to model verification. Although finite state verification techniques like model checking are not as general as theorem proving based verification approaches in terms of kinds of properties that can be proved, model checking is guaranteed to terminate and requires less mathematical sophistication [43]. In model checking, the specification properties a system is required to meet are expressed as formulae in some branching or linear temporal logic. With these temporal logic specifications, a model checker can report errors and provide useful clues as to where faults occur. These are usually in the form of counterexamples, linear paths leading to a violation of the formulae in the model.

Unfortunately, it has been long established in its respective field, that model checking is solely for verification purposes and does not recommend or apply any possible modifications to the system. To this point, there exist approaches posed ten years ago which provide possible modifications satisfying user defined behaviour, though do not do so based on temporal reasoning or have any generalised framework. Recently in 2008, Zhang and Ding proposed a generalised framework for modification of Kripke Structures over CTL specifications which recommends possible candidate updates for finite state transition systems [101]. Although the approach proposed by Zhang provided updates for finite models, inherent complexity and the model explosion problem made it infeasible to scale for industrial applications.

The method we implement utilises counterexamples based on the knowledge that they represent a minimal intelligible subset of the model leading to property violation, thus giving a reduced model space for update at the cost of generating tree-like counterexamples and restriction to universal quantifiers in CTL (ACTL) to express model specifications. Local model update effectively localises repair to sections of the model where the violation occurs. To demonstrate the problem posed by complex specifications and the utility provided by this approach, we consider an example of localised system update on a classic example in model checking, known as the microwave oven model.

1.1.1 The Microwave Oven Model

A good preliminary example demonstrating ACTL local model update is the microwave oven case study, which was originally devised in [24]. This example illustrates how counterexamples can be used to isolate fault regions in a hardware model and repaired to reintegrate a fix into the original model. Consider the scenario where we wish to verify the relative safety of a simple piece of consumer electronics. To do this, we would devise a finite state transition model representing all the states the device can be in and what operations executed cause the system to change state.

Here, we have a model which represents the operation of an abstract microwave, with operations that allow *starting*, *opening* the microwave, *heating food* and registering an *error* state. The relationship between the combinations of states is represented in Figure 1.1.

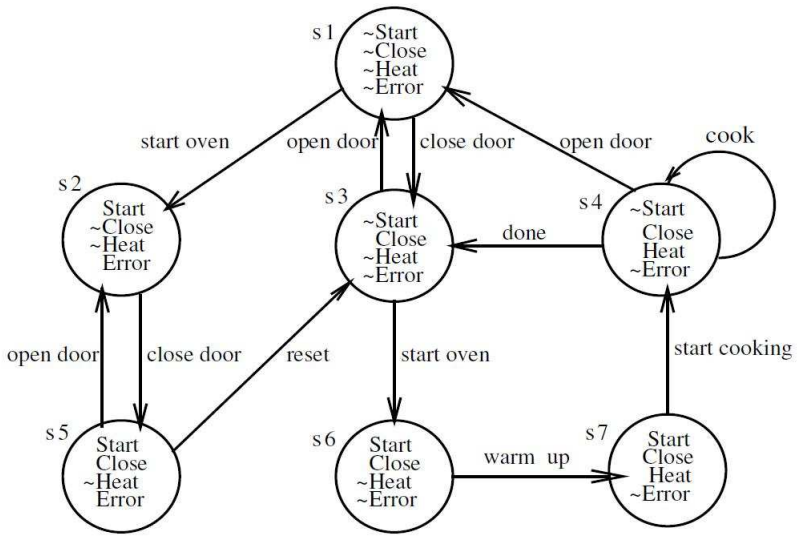


Figure 1.1: Transition graph of a microwave oven.

Here, there are provisions made for the door to open and close; attempting to start the system with the door open will cause an error state to occur, but can be reset once the door is closed. The natural usage of the microwave is emulated by closing the door, starting the microwave, heating food, stopping the system, turning off heat and optionally opening the door. This results in the creation of seven reachable states, $S = \{s_1,\ s_2,\ s_3,\ s_4,\ s_5,\ s_6,\ s_7\}$.

The condition we want to guarantee for this model is that from every state of the system, if the microwave is started it implies that in all possible futures heat will be applied. We can represent this specification in ACTL as AG($start \rightarrow$ AF($heat$)). To find out if the system satisfies this condition, we need to locate an example of where the system is in the *start* state, but *heat* is not applied at some given future point. In ACTL the witness to this counterexample will satisfy the complementary formula EG($start \wedge$ EF$\neg heat$). Applying model checking, we find linear counterexamples to the initially described property, the infinite path π witnessed by $[s_1,\ s_2,\ s_5,\ s_3,\ s_1,\ \ldots]$. We see this fault is caused by *starting* the microwave while the door is open, causing an *error* and never allowing *heat* to become true.

In earlier approaches to model update proposed by Ding in [42], the algorithm required consideration to every state in the model, states were checked for their ability to determine if any state in the model satisfied the converse of the property to locate the error source. From here, different primitive updates could be applied in combination to solve the issue. Possible updates include removing the relation between s_1 and s_2, removing the states s_2 and s_5 and replacing s_2 or s_5 such that they do not violate the property. This gives a total of five possible minimal updates, two of which could be considered dangerous from the perspective of the developer, modifying s_2 or s_5 to satisfy the proposition *heat*.

We can see that this technique is not complete. As counterexamples are local regions representing the subset of states of the model which do not satisfy the property, we can use these counterexamples as a means of error localisation. Further to this, a means of formally describing domain based information pertinent to update is required. From this, we apply updates to the counterexample we generated in model checking and can define a variable constraint automata dictating that if *heat* becomes true when the *error* label is true, we define this as an unacceptable state for the system to be in. This would then put the automata into a violation trap state.

Considering the counterexample, we can see there exist *four* states to apply the update. Using the property, we are guided by the semantics of AG to check the four states for states where *start* is true and there is no future state where *heat* becomes true. This is true for s_2 and s_5. Applying minimal change for counterexamples we see the possible changes involve removing the relation between s_1 and s_2, removing s_2 and s_5 and replacing s_5 with such the new state satisfies *heat*. The last update is removed from the update space as it causes the designed automata to transition to a trap state. From this simple example we can see that local model update with constraints allows a much more intuitive and efficient means of deriving an update to the model such that it satisfies the property and adheres to necessary developer constraints.

1.1.2 Motivation

With model checking theory being a matured field of research, a foundation exists in the form of highly optimised model checking tools which generate linear counterexamples. These can be used to guide repair, as initially discussed in [10]. There is currently no existing universal model update tools for guiding and localising repair through counterexamples. Further to this, there are limitations on directing the model update system in [101] towards permissible updates which adhere to action and variable related constraints inherent in the system, but not explicitly protected in update. Devising constraint automata which dictate what are acceptable states and actions within the model can simplify the update selection process and allow more context appropriate updates to be generated.

Based on the research performed, we integrate model checking and counterexample generation methods with model update techniques in an attempt to close the gap between functional generalities and computational effectiveness. The technique uses specifications expressed in universal computational tree logic on counterexamples generated in model checking sessions to localise update. Further to this, we extend local model update with constraint automata to allow better control over types of updates which are allowable and to improve efficiency. The major contributions presented in this thesis are discussed in the following section.

There is an enormous wealth of research into system verification and automated repair; in this thesis we make the case for an extension on model update techniques that is more accurate and efficient than those previously posed. We will review the past literature present for model checking theory and its applications, with a focus on the NuSMV verification environment, analyse previous attempts in system debugging and repair, and discuss recent developments with model update. To begin with, we give a history of model checking and give insight into the technique.

1.2 Background

1.2.1 Model Checking

Model checking is an automatic technology which addresses the following problem: given an abstracted finite model M of some system, test automatically whether this model satisfies a given specification ϕ [24]. Model checking was originally devised independently by J. Quielle and J.Sifakis in [82], E. Clarke, and E. A. Emerson in [21, 25, 45][1]. A full history of the thirty years of model checking and the surrounding topics can be found in [3, 24, 66].

This technique is useful for both hardware and software systems to determine if a finite representation of the system holds the required properties. These may include properties which detect liveness between processes sharing some resource, or absence of deadlock between interacting processes in some model. With this in mind, model checking is designed for finite state systems, often containing concurrent behaviour and most usually are reactive systems. Verification is performed in an automatic fashion, such that a binary response as to the satisfaction of the model is returned.

In the underlying theory of model checking, we express the problem as the given model M, at a specific state s (often the initial state) needs to satisfy the formula ϕ in some temporal logic. This is written formally as $(M, s) \models \phi$.

A fundamental issue for model checking is devising a natural abstraction of the system S from the hardware or software in question, such that it represents a finite model which can be checked for correctness by some property. Similarly, expressing the system property required in the model by the developer in temporal logic in such a way that will find a system flaw (if present) is another major issue for the process. Also, devising algorithms and data structures which allow for the handling of large scale search spaces is another consideration which determines efficiency in model checking.

[1] A cited algorithm for model checking for the propositional branching time logic CTL was presented at the 1983 POPL conference [26].

It should also be noted, in general cases hardware systems are finite, but software systems are often infinite systems. Abstraction techniques use inductive methods to prove a specification over an infinite domain in some finite number of steps with theorem proving, which may work for structural components but fail at a system level.

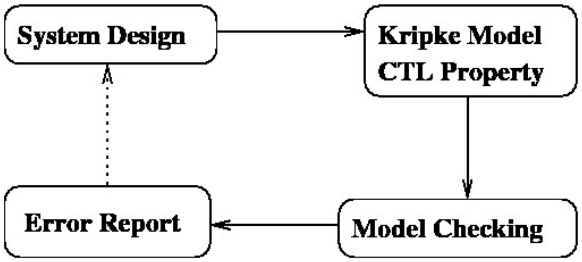


Figure 1.2: Previous model checking methodology.

The traditional method of model checking is displayed in Figure 1.2 and Figure 1.3. Here, some system design is translated through an abstraction mapping to a finite state Kripke structure and a corresponding temporal property is devised to determine if the model holds the property.

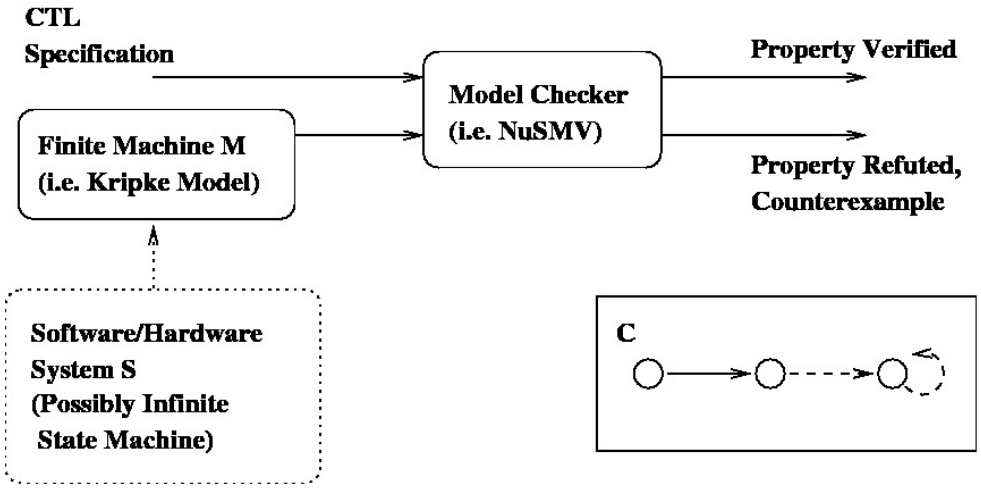


Figure 1.3: Process for counterexample derivation.

These are passed as arguments to the model checking tool which exhaustively searches the state space to find some instance of property violation. With this, any instance of design flaws discovered can be noted for redesign to improve or taylor the previous design to some specific use. Traditionally, model checking is a post design

method of catching system flaws before a commercially used system is implemented. In ACTL design flaws come in the form of counterexamples, linear paths leading to the specific violation.

The finite state property of models should be emphasised as a necessary requirement for the model checking process. Model checking exhaustively searches the state space via case analysis and based on the property returns any present design bugs. The finite model property for system models also guarantees termination of the model checker based on the models finite size. These finite models can be visualised as transition graphs, where edges are transitions between states and vertices are the state sets of labels describing the finite models current position.

Model checking generates a binary outcome from the checking session, returning true if the model satisfies the property and false otherwise. It is possible, based on the property verified, to return additional information on the checking session. If the property is translatable to an ECTL property and is satisfied in the model, a witness trace can be returned, explaining how the model satisfies the property. If the property can be expressed in ACTL and the property is unsatisfied in the model, a counterexample can be extracted explaining the violation in the model. This thesis focuses on the latter, using counterexamples as a means to localise the region of modification.

Model checking has a history of application to hardware and software systems represented as finite transition systems. In [104], Zhao et al. applied the technique to the Sliding Window Protocol, used in TCP/IP, and tested protocol properties such as data integrity, liveness and information consistency. Included in the modelling is intruder modules to illustrate the link between illegal data modification and violation of protocol properties such as data integrity[2]. A more exhaustive study of the rich history of model checking and related technologies can be found at [25, 46, 66] and throughout this thesis many applications of model checking for system verification will be presented.

[2]Fokkink et al. in [48] also verified the sliding window protocol independently using the μCRL process algebra toolkit.

ACTL and Specification Logics

For the purpose of this thesis, we use ACTL to specify system properties. ACTL is a branching time logic which employs a tree-structure as a means of modelling time. With this representation of time, we can unroll states in a finite state transition system to represent it as an infinite parse tree. This allows future paths that bifurcate and, when executed, a set of paths realised.

Semantics for ACTL are determined in relation to Kripke Structures. Syntax of ACTL can be defined inductively using Backus Naur Form:

$$\phi ::= \top \mid \bot \mid p \mid \neg p \mid (\phi \wedge \psi) \mid (\phi \vee \psi) \mid (p \rightarrow \psi) \mid \mathrm{AX}(\phi) \mid \mathrm{AG}(\phi) \mid \mathrm{AF}(\phi) \mid \mathrm{A}[\phi \mathrm{U} \psi]$$

In ACTL, the universal path quantifier allows for determining satisfaction of all paths from a state by some temporal query, ACTL has no notion of single path quantification.

Other temporal logics which are applicable in model checking are CTL, which contains existential quantifiers over paths and LTL, where semantics are defined over individual computational paths and have no notion of branching path quantification. Extending CTL is CTL*, which applies nested modalities and boolean connectives prior to applying any path quantification.

1.2.2 The State Explosion Problem

A well known issue in model checking is the state explosion problem. The large state count of system models can be accounted for in many ways, but in general is put down to the inherent detail required to fully represent the subtleties of the design of a system. Concurrency is one of the prime causal agents of system complexity; explicit state system verification requires that each individual components state count be combined against each state of the other running components to get an accurate portrayal of the possible states the system as a whole can represent. As can be seen in industrial cases of system verification, it is often the case that countably large numbers of components exist. This combination of components state space causes state count explosion, making runtime of verification algorithms in worse case scenarios intractable without optimisation.

Many methods have been proposed for alleviating the problem of state explosion. Popular methods include selecting subsets of ways of interleaving executed transitions such that they only represent possible interleavings of modules in the system, usually by means of system constraints [3, 24]. The use of binary decision diagrams (BDDs) was pioneered by McMillan in [76]. Another favourable approach to handling the state explosion problem is bounded model checking. Clarke et al. provides a sound analysis of the bounded model checking approach in [22].

Other successful approaches to state explosion handling includes SAT-based optimisation in symbolic model checking. Here, model checking can be done in an unbounded fashion as a method of alleviating issues of memory overflow found in BDD approaches to model representation. Conjunctive normal form is employed as a means of representing states and relations in a model checking session and SAT methods can be used to perform checking functionality. An approach to this was outlined by Kang and Park in [69]. In this article, Kang and Park show the boolean satisfiability approach verified more system circuits than previously posed BDD-based symbolic model checking approaches such as the one devised by McMillan. Another approach to handling system complexity was posed by Demri et al. in [36] called parameterized complexity handling, where synchronised components are taken as parameter for non-flat systems. In the following section, we will review the major approaches to ameliorating the issues of state explosion: Binary Decision Diagrams and Abstraction.

Binary Decision Diagrams

One method to alleviate the state explosion problem was the introduction of symbolic model checking, where binary decision diagrams were enlisted as a method of representation of models as formulas in the modal μ-calculus. The first conception of this was proposed in [12] by J. R. Burch et al. In this approach, the μ-calculus is used as a specification language, such that the relations and formulas of system models can be represented symbolically. This representation exploits regularities in system design, often based around concurrent behaviour.

In model checking, binary decision diagrams (BDDs) represent boolean func-

tions. These boolean functions can be used to represent states and transition relations in model checking, such that the model representation incorporates sets of states over individual states. This allows a symbolic model checking implementation. BDDs can be represented visually using the construct of binary decision trees. Non-terminal nodes are represented by variables labelled a_0, a_1, a_2 and terminal nodes are labelled a binary evaluation 0 or 1. With this, each binary decision tree represents a unique boolean function. However, binary decision diagrams are largely inefficient representations and contain many redundancies which can be removed. Variables can be given an ordering, such that each BDD is unique and equivalent boolean functions map to a unique BDD. This type of BDD is called an *ordered binary decision diagram*, or an OBDD. The first implementation of OBDDs was in Ken McMillan's thesis for SMV, where OBDDs are used as a structural method of alleviating the state explosion problem and reduce running time [76]. OBDDs can be generated through the removal of duplicate terminals, non-terminals and by merging nodes and performing redundancy tests [66]. Using this method, it can be determined if two BDDs are equivalent post optimisation. Given two reduced OBDDs with a compatible variable ordering B_1 and B_2, these OBDDs will have an identical structure if they represent the same boolean function.

Abstraction

In model checking, abstraction is the process of deriving a simplified relational model from a more complex system model. Abstraction allows us to remove details of the model which are irrelevant to the property, with the goal of simplifying the model and thus reducing the search space in the verification process [66]. Abstraction was introduced as a means of reducing model complexity as a complement to OBDDs scalability with complex applications [17]. An example of an abstraction is the temporal logic property used in checking against the original system model; model checking is the process of mapping the temporal property against the system model and looking for cases where the abstraction does not map to the model.

Manual abstraction is a common task undertaken by system developers, and is an important process in designing a finite model from the original hardware or software system. Many automatic methods of abstraction have been designed over the

years, but many do not have any formal justification[3]. Another important process related to abstraction used in verification and automated repair is *refinement*, the process of using the higher level specification to generate a more complex lower level implementation which still satisfies the original specification, but also includes more detail or required specification. In this sense, the refinement process creates a more realistic model of what it is representing and is considered more *concrete*.

A central text for abstraction in model checking is *Model checking and Abstraction* by Clarke et al. in [26]. This paper set forth an early implementation of abstraction in model checking, which approximates canonical abstractions such that they can be symbolically executed. The system devised used properties expressed in the CTL* branching time logic and showed the effectiveness of the approach by verifying a pipelined ALU circuit containing 10^{1300} states. Following this, an approach to abstraction refinement was posed in [23, 28] by Clarke et al. which generates iterative abstract models to the goal of verifying an abstract model of the concrete system. This proposed system generates an initial abstraction by analysing control flow of the original model. From here, abstract models are verified for property satisfaction. Abstract models may admit spurious counterexamples which can give useful information as to the correctness of the abstraction. Based on this, the model can be refined and the process iterated to verify the correctness of the concrete model. This approach is demonstrated with the created tool *aSMV*, an extension on NuSMV, on a *Fujitsu IP* core design with $10,000$ lines of SMV code.

Another early approach by Wing and Vaziri in [98] highlights an early attempt at abstraction of complex systems to exploit the nature of the system and the property to be verified. Wing and Vaziri made it known that there is no formal justification behind the abstraction function and it is the role of the developer to determine these functions manually. In this paper, they demonstrate application of abstractions applied to models by verifying abstractions of three cache coherence protocols, the Andrew File Systems $AFS0$, $AFS1$, and $AFS2$[4].

In [47], Felfernig et al. gave a scheme for using hierarchical abstractions to alleviate complexity in configurator knowledge bases in model based diagnosis. These

[3]Examples of this can be found in [23, 98].

[4]AFS0 and AFS1 protocols are also used in case studies for model update analysis by Zhang and Ding in [101].

configurator knowledge bases were used in product configuration applications where new component types or regulations need to be adhered to and products can be reconfigured. In the paper, algorithms are given for the hierarchical abstraction methods, to the point of developing an industrial constraint based configurator library. In [74], Lawesson et al. introduced a fault isolation system from model checking on concurrent systems. This application used automatic abstraction methods based on observational equivalence to handle state explosion and static analysis, to the point of creating a total function from message logs to show where faults are located. This allowed the author to reduce the fault isolation to a table lookup, such that tables could be used to find non-diagnosable system failures and redundant error messages.

Another approach was put forward by Boyer and Sighireanu in [9]. Abstraction methods were utilised to simplify sources of complexity in the *Pragmatic General Multicast Protocol* for the verification of reliability specifications. From these abstractions, a formal model was generated and constraints to apply to parameters were obtained. A method for archiving successful abstraction functions generated in XML for PROMELA programs was put forward in [35]. XML was used as a translation language between the PROMELA programs and the higher level abstractions generated. This had the benefit of allowing many highly optimised XML tools to be used for generating abstractions. This paper also introduces the use of the implemented tool $\alpha SPIN$ for this reason. In [17], Chauhan et al. put forward two methods of model abstraction analysis using SAT checking. Firstly, a method where variables irrelevant to the checked property were made invisible and set as input variables is proposed. Further to this, image computation is used as a means of pre-quantification of variables, to allow BDD model checking to be performed on the system. NuSMV was used to check for counterexamples. If counterexamples were found, they were simulated in the concrete system with a SAT checker to determine if they were spurious.

Building abstractions manually and automatically for model checking purposes is a large field of research with entire dissertations dedicated to it. For a deeper analysis into the concepts of abstraction refinement, the reader is directed to *Abstraction Refinement for Large Scale Model Checking* by Wang et al. [96].

1.2.3 Tools and Implementations

Model checking has been fruitful in more than the theoretical arena, as over the years many model checking tools have been created and have found use in industrial applications ranging from distributed cache coherence protocols, to verifying robotics controllers. In this thesis, we focus on the NuSMV verification suite [15, 19]. NuSMV was developed by E. Clarke, A. Cimatti, F. Guinchiglia and M. Roveri between Italy and Carnegie Mellon University in the United States [20].

The SMV model checker was developed by Ken McMillan in 1992 [76]. SMV was based on binary decision diagrams as a means of optimisation of the checking process and is an early example of branching time logic being used as a means of model checking specification. The Cadence SMV model checker is another offshoot of the SMV technology, developed by Ken McMillan with Cadence Berkley Labs. Cadence SMV was developed with compositional systems in mind, such that infinite systems can be represented using a compositional model, and subsequently verified.

Other than the SMV branch of model checkers, which initially focused on BDD based model checking with branching time specifications, there also existed model checkers which were developed to handle different types of specifications and model types. The model checker SPIN was developed primarily through G. J. Holzmann at Bell Labs and was created for simulation and verification of distributed algorithms [46, 62, 63]. SPIN uses the Promela language to specify system behaviour in LTL in a programmatic manner similar to C. Related to SPIN is Microsoft's SLAM toolkit [5, 64], which allows verification of safety properties for system software written in C without user abstractions or specifications. SLAM provides the tools for abstraction of C programs, model checking over boolean programs and the ability to discover further predicates for boolean program refinement.

Some other model checking technologies which have found use in the academic field include the UPPAAL checker for verifying dynamic properties in real time systems [73], Mur-ϕ [38], generated by Dill et al. at Stanford University for industrial hardware verification, Hytech [61] for linear hybrid automata[5], Kronos [31] for real-

[5]Developed by Cornell University in 1996 [1].

time systems by VERIMAG, and DESIGN/CPN for *Coloured Petri Nets* [67] by the Meta Software Corp. Further to these, some model checking technologies which deserve further review include MCK, FDR, COSPAN, Concurrency Workbench, Mex, and EXP. While this is not a comprehensive list, as the number of academic model checkers increases yearly based on new technologies and approaches, this covers many of the well known tools.

In [43], Dwyer et al. proposed a verification system for concurrent Ada programs called FLAVERS (FLow Analysis for VERification of Systems). Ada tasks are analysed for verification based on operator defined behavioural properties over event driven models. In this framework, FLAVERS generates abstractions of the system as a whole, known as *Trace-Flow Graphs* (TFGs) and individual tasks called *Control-Flow Graphs* (CFGs), which are generated and refined. Combining with the TFGs, these together are model checked. This system also allows the use of user defined constraints.

In the following section, we will give focus to the model checking tool SMV, which is used in generating the implementation to the theoretical system posed in this thesis.

NuSMV - The Symbolic Model Verifier

NuSMV is a model checker, considered one of the benchmarks for verification of finite system models [20, 24]. NuSMV was developed by E. Clarke, A. Cimatti, F. Guinchiglia and M. Roveri between Italy and Carnegie Mellon University in the United States. NuSMV is an extension on SMV which allows added functionality, using SAT and BDD based methods of symbolic model checking (NuSMV 2).

NuSMV extends the functionality of SMV in that it allows the definition of process modules, LTL specifications and other functionality not previously offered. NuSMV processes the represented model and returns true if the model satisfies the specifications and, if false, returns an error trace explaining where the violation occurs. NuSMV also allows for the representation of LTL, various forms of constraints and the definition of processes by the developer.

SMV is also the name of the language used for representation of the system

models checked by the implemented system. SMV is used as the input language for the implementations SMV, NuSMV and Cadence SMV and allows a modular and expressive representation.

An SMV program description consists of one or more modules which accept arguments and declare variables local to the specific module. Each module in an SMV program description requires a label be assigned to it. Of the modules present, there must be a module labelled *main* where the program executes from. In each SMV module a *VAR* declaration section demarcates where variables used in the module are assigned types. *ASSIGN* declarations mark where the transition relation for the underlying model is defined, by initially assigning values to variables using the *init()* call (*e.g.* a boolean variable *var* can be initialised as *init(var) := 0*).

After this, a *next()* call assigns a new value to each variable at each transitional iteration, *i.e.* it dictates the transitions that can occur in the underlying finite model and the type of values that can be assigned to a given variable in the models states. Conditional case statements can be applied to next calls, such that present system conditions can be referenced to direct transition relations and model the desired behaviour. Specifications declared in SMV are prefixed with the tag *SPEC* and follow the syntax of CTL specifications (NuSMV also allows LTL specifications and simple constraints such as fairness). To demonstrate NuSMV syntax, we present an example of an *asynchronous three bit counter* with a specification of desired model behaviour[6] in Figure 1.4.

We define an inverter module that takes as argument a single binary input and assigns its output value to the negation of the input. In the main module, three inverter objects are defined and are chained, such that the output of one is passed to the input of another in a circular fashion. To complete the circular chain, the final output of *gate3* is passed to *gate1*.

The property we want the represented system to hold is that at all states on all computational paths, all future paths should lead to both possible values for *gate1.output*, *i.e. gate1* should oscillate indefinitely and, by implication, so should *gate2* and *gate3*.

[6]This example can be accessed at the NuSMV examples collection at http://nusmv.fbk.eu/examples/examples.html, the reader is referred to [15] for further NuSMV usage information.

```
01:  MODULE inverter(input)
02:    VAR
03:      output : boolean;

04:    ASSIGN
05:      init(output) := 0;
06:      next(output) := !input;

07:  MODULE main
08:    VAR
09:      gate1 : process inverter(gate3.output);
10:      gate2 : process inverter(gate1.output);
11:      gate3 : process inverter(gate2.output);

12:    SPEC
13:      (AG(AF(gate1.output))) & (AG(AF(¬gate1.output)))
14:        /*A declared specification phi*/

15:    FAIRNESS running
```

Figure 1.4: An asynchronous three bit counter in the extended SMV specification language.

1.2.4 Counterexamples

Highly regarded as a tool for error diagnosis, counterexamples are a useful product of model checking sessions, primarily in a design context where engineers and designers require an example of where a system fails by some given property. The ability to exhibit cases where a model explicitly fails some specification can save a great deal of time in the design cycle, over just being notified that the model fails by some desired property. Counterexamples are an effective feature to convince an engineer of the value of formal verification for the localisation and correction of system bugs in some design.

In [28], Clarke and Veith provided a survey of the contributions to the study of counterexamples. The article contained an analysis of the generation of counterexamples, the state explosion problem, the application of counterexamples to abstraction refinement and the need for abstracting out irrelevant variables in counterexamples to facilitate better understanding of where the violation occurs.

If some model checker finds an ACTL formula ϕ does not hold for some $s \in S$, a counterexample C can be found which explains the violation. A property ϕ claimed to hold for each element s in a model M can be disproved by displaying an instance of $s \in S$ such that ϕ does not hold true for S, and s is reachable from the initial state.

Counterexamples can be said to have the property of explaining the violation of ϕ in a rigorous manner in the model M, using C. Clarke et al. gave a set of expectations of a found counterexample in [27]. Counterexamples:

1. *are a subset of the states of some model;*
2. *violate a property. C violates ϕ, $C \not\models \phi$;*
3. *explains the violation of ϕ in M with C;*
4. *counterexample C is viable.*

Another important note is that with counterexamples, it is equivalent to say $C \models \neg\phi$, in this way the counterexample can be seen to *witness* $\neg\phi$. Saying this, existential properties cannot be disproved by counterexamples, although they can be proved in the case they hold true by generating witnesses [27]. Negating some ACTL formula ϕ yields its ECTL equivalent $\neg\phi$.

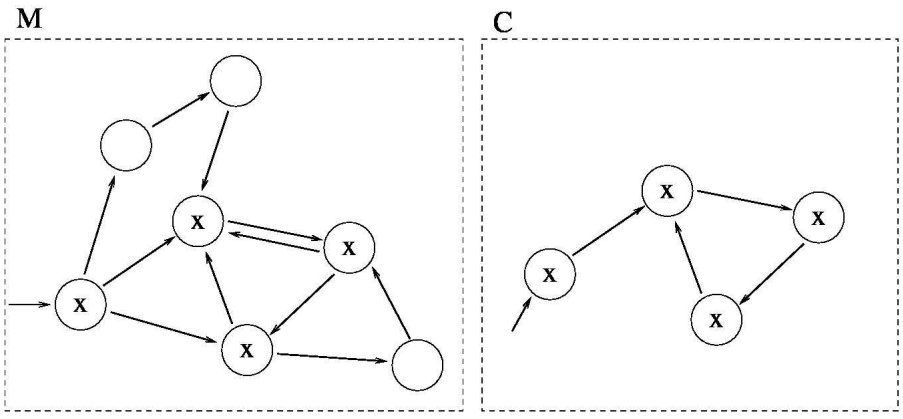


Figure 1.5: A linear counterexample of AF$\neg x$ [27].

Example 1.1. *The ACTL specification $AF\neg x$ denotes "On all paths, x is not true at some point in the future". If the specification $AF\neg x$ is violated then there is some infinite path in M where x always holds. This is referred to as a counterexample of $AF\neg x$. Figure 1.5 illustrates a case where some Kripke structure M violates the property $AF\neg x$. Counterexample C describes the infinite path witnessing the specifications existential converse EGx; the violation on C is an explanation of the violation on M.*

1.2.5 Model Update

An extension of model checking functionality, the problem of model update poses a simple question: given a Kripke structure of some abstracted system and a temporal property formula found unsatisfied, what is the smallest set of atomic modifications that can be performed on the Kripke structure such that it will satisfy the given temporal property. This approach has its roots in the combination of knowledge update and program debugging and repair[7]. In [6], Baral and Zhang discussed knowledge update and minimal change in knowledge domains between worlds. This work is based on the modal logic S5 and was the first integration between model checking and knowledge update, leading to the creation of model update.

In [42, 101], a generalized framework for model update was proposed. Here, some Kripke structure M not satisfying an arbitrary property ϕ ($M \not\models \phi$), and expressed in Computational Tree Logic (CTL) could be modified based on a set of steps of primitive updates operations. This was coupled with minimal change principles, such that the modifications lead to the creation of the model M', satisfying the original property ($M' \models \phi$) on the condition that there is no other set of atomic modifications on a model M'', such that $|M''| \leq |M'|$. We could then utilise M' as a candidate solution to repair the original program. In this system, five primitive update operations were allowed, including removing or adding an isolated state, removing or adding a transition relation or changing the labeling function on a set state. With these and the minimal change criteria, the framework was set for a model update system.

[7]See [6, 56, 87, 99].

While this is a legitimate method for modification of simple models, difficulties arise when presented with models of larger scale industrial complexity. Model update utilises the entire Kripke structure and the space of possible models which both satisfy the required property and is considered equally minimal explodes for models of any significant size; this is known as the model explosion problem [39]. Another known issue is that when given two Kripke structures, determining if one is an admissible model update of the other is co-NP-complete, even for simple properties [101]. To compound this, there are no criteria that dictates which primitive updates are more preferable between modifications, whether adding a state or removing a transition to enact satisfaction.

Zhang and Ding extended the update methodology to better handle the model explosion problem in [39, 101]. Here, a new update principle called minimal change with maximal reachable states was used to optimize the search for candidate updates, such that the pool of committed models were reduced to a subset of fewer strongly committed models. With this framework, a focus was placed on updates which maintain prior reachability between states, initially founded to highlight updates which changes the behaviour of the system in a minimal way without effecting state reachability.

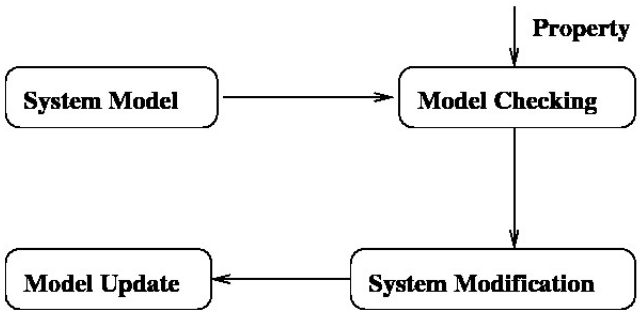


Figure 1.6: Previous model update methodology.

Another parallel approach to model update is model revision. As model update is a technique for maintaining consistency of dynamic systems by desired properties, model revision applies to a static context. In [58], Guerra et al. devised an approach towards model revision based on AGM belief revision postulates, defining a revision operator and characterisations drawn from revision theory.

Implementations

Zhang and Ding's approach utilized SMV and taylored state reachabilty algorithms to effect update, details of algorithms used in model update are analysed in [40]. The approach was implemented in the C language and used explicit state Kripke structures encoded in C, OBDD's were not considered as a means of optimisation, based on the constraints of research time. A case study demonstrating the application of this approach can be found in [41] and details giving semantics and implementation of the model update system can be found in [42].

In [34], Menezes et al. developed an alternative method to Zhang and Ding based on the methods produced in [80], extending model update to modification on partial models represented by labelled transition systems. The proposed system used α-CTL for specifying properties corresponding to actions in a labelled transition system. The system used modified primitive update operations, such that addition and removal of transitions induced by actions would occur on given partial models. Further to this, the minimal change criteria were altered to take into account the action based primitive updates, only taking into account states and transitions and no altered label functions. Characterisations for the action based update procedure were also provided.

Another alternative implementation of the model update framework was posed by Cacovean and Stoica in [14]. This implementation used the ANTLR toolkit, a framework for developing concurrent and real time systems for the representation of the update architecture. The model update framework was represented as an algebraic compiler which returned candidate modified models from system specifications, given as argument. In their paper, the technique was demonstrated by updating a finite model representation of the elevator controller system used in many model checking case studies[8].

[8]Preliminary work by Cacovean et al. on developing an implemented universal model update compiler can be found at [13].

1.3 Related Work

Counterexamples

In [11], Buccafurri et al. analysed the complexities of linear counterexamples derived from ACTL model checking sessions. The research found there is no simple characterisations of ACTL formulas that generate linear counterexamples. Further, Buccafurri determined the maximal set LIN of ACTL formula templates, whose instances are generated by substituting atoms with arbitrary pure state formulas, guarantees linear counterexamples. Algorithms were proposed for Kripke structures witnessing the failure over a single path, which could be computed in polynomial time. In [4], Ball et al. comments on the research developments in counterexample generation and put forth a method for generating multiple simple counterexamples exhibiting independent causes. The approach was undertaken for C programs in the SLAM environment. The technique involved finding transitions in the error trace that did not appear in a correct trace. Meolic et al. proposed a method for guaranteeing linear witnesses and counterexamples for the action based temporal branching logic in [77]. The approach introduced witness and counterexample automata, to the point of recognition of linear counterexamples.

In [44], Clarke et al. analysed how counterexamples and witnesses were generated for CTL and CTL* properties. The author worked in the framework of symbolic model checking for OBDDs using SMV as the verification tool. In [85], Shen et al. devised a method for counterexample minimization for ACTL property *loop-like* and *path-like* counterexamples. System proposed used a notion of guided cubes over states, represented as BDD's to remove irrelevant variables from the counterexample description. Experiments were run in the NuSMV environment to demonstrate the efficacy of the approach.

In [97], Wang et al. provide an optimisation to existing abstraction refinement techniques for industrial scope sequential circuits by analysing abstract counterexamples. This technique was implemented by breaking down combinatorial logic cones with boolean network variables and treating these and state variables as

atoms in abstraction. A refinement algorithm was posed, using concepts of ariadne's bundle techniques and synchronous onion rings. The approach scales, in that all analysis and computation in the refinement algorithm occurs on the abstract models. Experimental results are available in the referenced article. Another approach to abstraction refinement based on counterexamples has been put forth in [86], by Shoham and Grumberg. In this approach, abstraction refinement occurs in a game based framework in CTL model checking using derived counterexamples in a 3-valued semantics. This 3-valued semantics refines abstractions by exploiting the understanding that verification sessions may return indeterminate results, giving cause for further abstraction refinement. Shoham and Grumberg put forward algorithms for the game based framework and give examples of application to show the efficacy of the technique.

In [52], Fraser and Wotawa analysed the problem of non-determinism in testing. A problem existed in non-deterministic models, where counterexamples generated from property violations can be spurious, based on committal to non-deterministic paths. The approach is based on generated test-case comparison with counterexamples of non-deterministic systems. The approach is demonstrated in NuSMV and an example is presented, demonstrating safety injection systems tested by LTL specifications.

In [94], Van Den Berg et al. discuss the development of a method of automatic interpretation for complex linear counterexamples derived from NuSMV sessions. The authors apply the approach to rail signal control tables and put forth two approaches: counterexample animation and natural language interpretation. Van Den Berg et al. also offered a method for counterexample interpretation based on providing domain specific details on variables, to better facilitate understanding for the end user. Further research has been done in using counterexamples as a means of error diagnosis. In the following section on error repair, we will look into the work done by Groce and Visser in [57] and analyse the effort towards counterexample information extraction.

As can be seen, much progress has been made over the last twenty years in counterexample research. Focus mainly centers on making counterexamples more legible to the developer, using counterexamples as a means of generating better sys-

tem abstractions, fault localisation, facilitating repair and updating counterexample generation techniques to make up for flaws or inbuilt assumptions. In this thesis, we will be using counterexamples as a means of localising faults to facilitate update, naturally ACTL is the temporal logic used for specification.

System Debugging and Repair

Automatic methods for programming debugging and repair is an area with a long history of research focus. The aim of automatic diagnosis is to identify the source(s) of fault and find some set of modifications that remedy the faulty behaviour. Error diagnosis in digital systems has a rich history reaching back to the 1970's [29, 33, 79, 83, 92] and over the years many systems have been proposed using different techniques and implemented in varying languages.

Looking back, early methods of error diagnosis started without any formal description of diagnosis for system design errors or formalised approaches. Devised techniques focused mainly on diagnosing hardware faults [91], in general were limited in expressiveness and temporal properties were not considered as a method of describing system behaviour. Methods focused on localised regions of the program code [68, 83] or presented specialised specifications to describe desired behaviour [32]. Theorem proving was also a popular method of fault detection and localisation. Error diagnosis occurred primarily on static states.

In [72], Kuehlmann et al. proposed an approach for automatic verification of transistor level circuits in CMOS design, using an error coverage algorithm for establishing equivalent boolean structure of circuits. Patterns which correspond to nets which likely cause errors were propagated from incorrect outputs backward into the circuit network. This is executed using a combination of BDD-based and simulation based techniques. This is was an example of a technique which did not apply model based methods.

One approach to error diagnosis was that of Model-based diagnosis (MBD) [88]. Model based diagnosis was defined as the search for behaviour unintended in the original program through the use of a model to describe the programs domain. With that, the model could be processed to determine bugs.

In most approaches to MBD, the model of the concrete system was expected to specify the semantics of the program executed in a general manner and provide observations which lead to expected output values; formal specification was not required external to the model given. This was appealing as an approach, as it allowed a description of the software to be automatically generated and used as a diagnostic tool.

Some leading approaches of model based diagnosis included the design of intelligent debugging systems, which discovered faults inherent in software and hardware designs. A branch of intelligent debugging was intelligent tutoring [92], where knowledge about particular student assignments and knowledge about common errors were used to assist students in debugging their code. This was designed to bootstrap the debugging mentality. Some examples include the *PROUST* environment for *PASCAL* and *Talus* for *Lisp*.

In *PROUST*, the intent of the student programmer was analysed by matching chosen programming elements against a plan library and uses an explicit model representation for the plan libraries. On the other hand, the tool *Talus* took a semantic approach, parsing the student code to a tree structure and matching it against referenced elements to provide code recognition, based on theorem proving techniques. Both systems operate on the existence of knowledge banks, representing deep understanding of the purpose of the given programs and languages in use. This however was a bottleneck, in that it required explicit external specification, and generally was only feasible to discover novice programming faults.

An early approach by Console et al. in [30] applied the model-based diagnosis approach to faults in software. The technique attempted to isolate faulty components represented in a logic programming environment by adding, removing or replacing logic clauses to find some configuration which returned correct output based on test queries and, when executed, can be suggested to the designer. Also presented in this paper was the notion of leading diagnoses to allow a sense of priority or causes which were more likely. An early thesis on automated program diagnosis by Shapiro [84] gave insight into approaches applied in prolog before model based diagnosis was formalised.

Early takes on formal model based system repair could be seen in frameworks such as [89, 90]. The system put forth by Stumptner and Wotawa focused on models represented in VHDL (*Very high speed integrated circuit Hardware Description Language*) for hardware design, similar to that used for concurrent system representation. In this framework a program diagnosis approach was taken where a fault was explained through some finite number of fault models. This system does not incorporate temporal properties for fault models, limiting its expressive capabilities. In [91], the VHDLDIAG tool was introduced as a tool for design support through model based reasoning for localisation of faults in hardware designs in the VHDL language.

Another similar approach to VHDL program fault localisation in large scale design was devised by Friedrich et al. in [54]. This approach used the VHDLDIAG tool and allowed for high levels of abstraction for handling programs of upwards of 10,000 lines of code.

Another solid example of hardware designs being modified was the work by Chung et al. [18] on VSLI (*Very Large Scale Integration*) circuits. This approach implemented a gate level design correction system for logic design errors and provided a tool for logic level correction. The technique applied boolean equation methods for locating circuit errors. Huang et al. proposed another error diagnosis approach for sequential circuits in the VLSI design environment [65]. In [78], Nayak and Walker proposed a simulation based approach for repair on combinational digital logic circuits. The technique proposed targeting small errors, based on circuit nets where minor changes are sufficient to effect correction, when new specifications were added or errors were inadvertently included in the design process. The approach could handle multiple errors on large circuits, based on simulation and symbolic algorithms. Another approach to design error diagnosis in combinational circuits was posed by Ubar in [93], four years later. The detailed approach localised faults in VLSI sub circuits and did not use a pre-specified error model. This was achieved through the use of back substitution heuristics to repair the fault on the sub-circuit, determined through verification techniques. The article put forth a technique for calculating the method in which the sub-circuit in question should be chosen for modification. This was explained to increase the efficiency of the process. In the same year, Veneris et al. published an incremental *DEDC* (*Design Error Diagnosis*

and Correction) approach [95] on combinational circuits in VLSI design. The technique used a simulation based debugging approach using test vectors to iteratively identify and correct design functionality, such that it brought the erroneous design closer to the desired specification.

Over the years, error diagnosis as a field of research has matured and as a consequence formalisms for determining faults and modifications have been devised. Model checking has evolved parallel with error diagnosis and many error diagnosis techniques have borrowed from or included model checking techniques to better locate and provide understanding of errors.

In 1992, Friedrich et al. [53] put forward a first attempt at formalising the repair process, extending on model-based diagnosis and concepts of temporal reasoning to describe inherent system purpose. Definitions are given to describe component failures in relation to actual, possible and plausible worlds and uses a framework of observations and actions performed on the system to apply repair of purpose on the model.

Later in [10], Buccafurri et al. introduced a formal approach integrating AI techniques with those of model checking to effect system repair. Abductive model revision techniques are used for modifications on concurrent computer systems with multiple processes. The system proposed the use of temporal logic operators in ACTL for representation of properties in the computer system. In this framework only transitional elements in the given Kripke structure were modified, no actual state changes occur in the system repair.

In [59, 60], Harris and Ryan used a similar approach, developing a tool that integrated system features into sections of existing software using McMillans SMV. The approach utilised notions of theory change and belief update to design a framework for novel feature integration in the SMV model checking language. This research generated a feature integration tool in the SMV environment, the SMV Feature Integrator, or SFI. In [60], they went on to prove a theorem showing the functionality SFI provided was an update operation. Earlier work showing the preliminary results of feature integration using *SMV* can be found in [81].

In [57], Groce and Visser devised an automatic method for extracting information out of multiple counterexample instances to better determine the cause of a fault. The proposed system used positive and negative sets of counterexamples generated from SLAM Java code to find counterexamples which exhibited the same error occurring at the error states. In [87], Staber et al. proposed a method for error localisation and system repair of sequential systems under the *Vis* model checking environment. The system used a game based strategy such that a winning strategy was implied found if there was a correction valid for all possible inputs of the system. Specifications could be expressed as LTL properties.

Another repair framework was implemented in *Vis* by Jobstmann et al. in their paper *Program Repair as a Game* [68]. In this paper, Jobstmann et al. theorised a game based framework for finite state models and automaton, representing LTL specifications. Finding a winning strategy corresponded to determining a repair for the finite model. This approach successfully used localisation techniques and proposed an optimal strategy, adding extra states needed to be avoided, and thus provided heuristics for devising memoryless strategies. Complexity was given as polynomial on the size of the model and exponential on the temporal property. A similar approach by Staber was given in [87], which used a similar game-based LTL approach on finite state machines for fault localisation and correction in sequential circuits. The framework set the games win state as when a correction was determined for all inputs, where a memoryless strategy was preferred. When a solution was found, components were replaced by new functions satisfying the property. In the paper, the technique was demonstrated on a locking algorithm and a minimal input function to promote better understanding. The system was also implemented with the *Vis* model checker.

In the 2006 paper [37], Dennis et al. put forward a theory for proof-directed debugging, proposing the use of proofs in functional programs in combination with other debugging and localising techniques (including counterexample generation), to locate bugs in a program. Proof planning was posed as a method of guiding the verification of the program and assist in the localisation process. Methods posed for proof planning include the use of loop invariants, induction schemes, corrective predicate construction and middle out reasoning. Highlighted in this paper was the lack of support by verification tools in general for identifying the cause of faults,

methods for fault localisation and recommended repairs, as also mentioned in [27].

Similarly again in 2006, Griesmayer et al. provided a method of fixing faults in C programs with boolean variables in a game based framework [56]. In this system, the protagonist picked some implementation for an incorrect program, if one could be found which did not require memory or stack contents, a repair on the boolean program was found. This system was completely symbolic, counterexample and game strategy based. The framework was applied to abstracted boolean programs produced from the Microsoft SLAM environment.

In [55], Gorogiannis and Ryan put forth a minimal refinement model for the addition of new specifications to existing designs. Using models in SMV, a system was proposed which allowed the addition of extra design specifications to a system expressed as transition systems and computational paths. Designs could be minimally refined in such a way as to not invalidate existing specifications, whilst satisfying required introduced behaviour expressed in universal computational tree and modal logic. To show system effectiveness, an example of mutual exclusion was demonstrated attempting refinements to the model by including further liveness properties.

From these methods, a pattern of formalisation of the repair process can be inferred; up to this point no generalised abstract framework for providing candidate modifications had been developed. All rely on specific programming languages and were limited in the sense of their use of temporal properties, constraint handling, complexity handling and strategies for performing the update.

1.4 Contributions

The central aim of research conducted in this thesis is to develop a computer-aided repair tool for system models, guided by counterexamples derived from model checking sessions, and extending previous update techniques. The desired outcome of this process is a corrected local model fix that can be reintegrated back into the original Kripke structure to affect satisfaction of the original system specification. The technique enlisted is illustrated in Figure 1.7. This differs from previous methods

of update, shown in Figure 1.2, in that only counterexamples are altered to satisfy the global model. In previous iterations the entire model was accepted as input, posing an infeasible search space for candidate fixes. In addition to this, our revised approach takes into account developer defined action and variable constraints through constraint automata, to better direct update such that the resultant fix better reflects required specifications.

The contributions of this thesis are as follows. We include the theoretical foundations for local model update and a prototype tool demonstrating local model update for simple counterexample fixes based on ACTL specifications. Extending this is the theory underpinning constraint automata guided model update that we use as a method for deriving updates based on constraints. Following the foundations, we provide algorithms for implementation considerations for localised model update and included action and variable constraint automata guidance. Finally, we generate a compiled prototype in the Python language which demonstrates tree-like local model updates on counterexamples generated from NuSMV model checking sessions. To demonstrate the applied approaches we include three sufficiently complex case studies in the following section. As discussed in the literature review for this dissertation, the author is aware of no other approach to performing model update which utilises counterexample traces as a means of localising specification violating behaviour in the represented system.

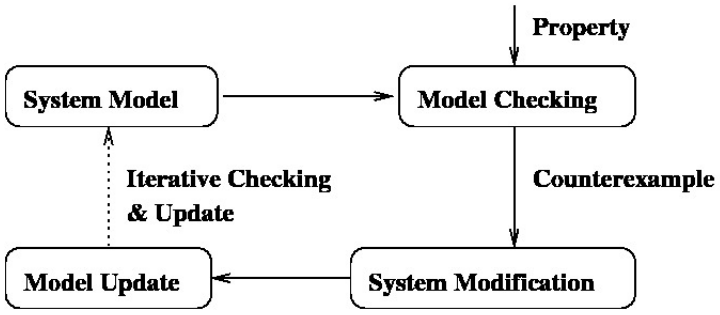


Figure 1.7: Proposed tree-like local model update approach.

1.4.1 Formalised Framework for Local Model Update

In the author's research, ACTL is the selected logic for describing behaviours of the hardware or software input model. ACTL is focused on for its ability to, in most cases, generate linear counterexamples which can be used to construct tree-like counterexamples [11, 27]. Further to this, software models in general are finite state machines and thus, to be represented, need a format that can be expressed in finite terms. ACTL formulas are interpreted as infinite computational trees derived from finite state transition systems, where each path in the tree describes a series of states.

In the theoretical portion of this thesis, we provide the formal underpinnings for the technique, including formula parsing and the semantic framework of ACTL over explicit state Kripke structures. We define a set of weak bisimulation based minimal change criteria, which gives us a more accurate definition for what is a smaller distance between tree-like models. This is necessary when trying to find an optimal update from the modification search space, based on what is possible that satisfies the property. We analyse the relationship of the approach to traditional belief update, providing proofs of the connection between the local update technique and the AGM update postulates first proposed in [70]. With minimal change properties and ACTL defined over Kripke structures, essential semantic characterisations are provided. These characterisations give update rules for each semantic satisfaction rule in ACTL in relation to models, such as AF and AG. This formalisation process allows us to develop a more efficient approach to local model update. From these characterisations the complexity of the approach for given cases is also studied.

With these characterisations and the structure of our temporal logic, algorithms are then designed for routing specific updates guaranteeing the most minimal changes to the localised updates are returned. ACTL local model update ensures that only the region pertaining to the fault is brought under consideration for update. This saves having to provide possible modifications which occur on a greater subsection and are more likely larger. We then implement our prototype ACTL system based on the theoretical approach and apply it to case studies to demonstrate its efficacy.

1.4.2 Case Studies

In this thesis, we apply the technique to some well known case studies in computer science to illustrate its use and demonstrate its application to real world domains. For this work we have selected three well known models which can be explicitly represented in the NuSMV modelling language. As summarized earlier, we apply the technique to the microwave oven Kripke structure, a hardware model repeatedly illustrated as a fundamental example for model checking. We present an application to the semaphore sharing model, a problem demonstrating semaphore deadlocks occurring between resources to demonstrate the update process on a sufficiently complex application. We apply the technique to a more complex case of the Sliding Window Protocol, used for buffering frame windows in TCP/IP, as demonstrated in [48, 102, 104]. We experiment on the protocol using varied window sizes to note how complexity effects the approach.

Finally, we give a demonstration of update in the SLAM environment on a modelled mutual exclusion problem, using variable constraint automata to define acceptable behaviour in determining candidate updated models.

1.4.3 Implementation of ACTL Tree-like Local Model Updater

This research introduces the use of *l-Up*, a software package developed for the generation of candidate fixes over NuSMV counterexamples. The implementation of this local model update system has been completed in the Python programming language. Python was chosen for its clarity in syntax, general philosophy of readability and pragmatism as to allow future developers of these techniques an easier learning curve. This will hopefully promote better integration of update methods within the model checking community.

Python is also known for being a high level language, is used for rapid prototyping and integrates elements of the object oriented, functional and imperative programming paradigms. Python has many applications in AI and has been gaining more ground in popularity recently over more traditional languages, based on

developer circles and companies that are hiring and seeking Python based skills.

C/C++ was also considered as a possible language but ultimately was not used. While C/C++ is known for its stability, speed and widespread use for model checking tools and system level implementations, the goal of this research project is the promotion of integration of counterexample and automata guided update and the generation of an initial prototype, which researchers can use out of the box and analyse. System models interpreted by the update system are not written in the Python language, but expressed using the NuSMV modelling language and are passed as argument to NuSMV for pre-verification. We analyse underlying system design written in the Python language in Chapter 4. Abstract algorithms developed in the earlier chapter and the design philosophy of model update are integrated into the designed local model update system.

To perform model checking and counterexample extraction the software utilises NuSMV as a backend. With these counterexamples, local update methods implement the derived theoretical results in the form of algorithms. This includes propositional update operations, characterisations based on weak bisimulation minimal change and search algorithms for finding the most minimal candidate change. The system returns atomic changes which satisfy the initial property. Algorithms are designed with modularity and future extension in mind. Bisimulation ordering and minimal change criteria are swappable such that other ordering heuristics can be applied.

1.4.4 Constraint Automata in Update

Although we have defined a method for optimising the application scope of model update, it is still the case that model update only takes into account properties expressed in temporal logic; we cannot direct the update process to only derive modifications which adhere to action or variable constraints expressed by the system designer. This could be useful, as a designer may require certain functionality to persist but the desired property still satisfied within the update region. For this reason, we extend the approach with the application of variable and action constraint automata, such that derived updates conform to detailed behaviour in the automata defined by the designer.

We provide the essential theoretical aspects of constraint automata, including automata definition for action and variable constraints and the semantics for automata over tree-like model paths. From these semantics, we extrapolate characterisations which can be used for implementing extension algorithms on local update algorithms and analyse the underlying complexity for performing update with constraints taken into account. Finally with algorithms defined, we show a case study which demonstrates the use of constraint automata on a modelled version of the mutual exclusion problem in Chapter 5, Section 5.3.

1.5 Organisation of Dissertation

The rest of this thesis is organised as follows. In this chapter we have given a background of model checking, system repair, diagnosis and model update and have given an overview of the proposed update technique. Chapter 2 provides the foundational syntax and semantics of ACTL and leads into the theoretical results of tree-like local model update, including minimal change with weak bisimulation, property persistence, the links between belief update and tree-like local model update, complexity results and theory of constraint automata compliance.

Chapter 3 gives the central algorithms for tree-like local model update, describing the primary concept guiding design from characterisations derived in the earlier chapter. In Chapter 4, we look at implementation specific details which provide the programmatic interface to allow local model update to function, such as formula parsing, SMV program variable domain extraction, elements fundamental to update and structures for constraint automata. Chapter 5 contains three well known case studies for tree-like local model update to demonstrate its applicability as a tool for system repair. Finally we conclude with Chapter 6, giving a summary of the dissertation and propose future directions for research into this field. Appendix A contains the user manual for the ACTL local model update prototype. Appendix B contains a detailed model of the *Gigamax Cache Coherence Protocol*, used as an example to show details of implementation. Appendix C contains counterexamples derived from model checking sessions used in the case studies, illustrating local model updates efficacy.

Chapter 2

Foundations of ACTL Tree-like Local Model Update

In this chapter we present the preliminary concepts supporting tree-like local model update, reviewing Kripke structures, the syntax and semantics of ACTL, and definitions for tree-like model update. We also give examples to clarify and reinforce the fundamental concepts of local model update. We then perform semantic characterisations, study the relationship between local model update and belief update and focus on satisfaction of persistence properties. We analyse computational properties, giving the main complexity result and some review of computing typical tree-like local model updates. We conclude the chapter by giving the theoretical aspects of introducing constraint automata as a means of better directing the update process towards modifications which comply to action and variable constraints.

2.1 ACTL Syntax and Semantics

To start, we review the syntax and semantics of ACTL and the background to local model update. Readers are referred to [24, 66, 101] for a deeper background into model checking, CTL and ACTL. ACTL is a fragment of Computation Tree Logic (CTL) that has attracted considerable studies from researchers, *e.g.* [11, 27, 77, 85]. CTL is a temporal logic formalism with the purpose of describing how states transition between one another in some reactive system [24].

Definition 2.1. *[103] Universal Computational Tree Logic (ACTL) has the following syntax given in Backus-Naur Form*[1]*:*
$\phi ::= \top \mid \bot \mid p \mid \neg p \mid (\phi \wedge \psi) \mid (\phi \vee \psi) \mid (p \rightarrow \psi) \mid AX(\phi) \mid AG(\phi) \mid AF(\phi) \mid A[\phi\ U\ \psi]$

where p is any propositional atom (variable).

Negation is restricted in application to propositional atoms (*i.e.* $\neg p$ is an ACTL formula if p is one). ACTL differs from CTL in that ACTL employs the use of only the universal path quantifier A and excludes usage of the existential branch quantifier E. A allows checks over all branches to assert the necessity of some condition be true over all paths from some state.

Besides Boolean connectives, ACTL provides linear time operators X meaning 'neXt state', F 'some Future state', G 'Globally true' and U 'Until'. Each time operator is coupled with the universal path quantifier to specify the scope the temporal token applies to [24]. It should be noted also AU is the only binary temporal operator.

Convention 2.1. *[24] Unary connective $\neg$, and temporal operators AX, AF, AG take precedence in parsing. Binary operators are next, in order of $\vee$ and $\wedge$, followed by $\rightarrow$ and AU.*

Definition 2.2. *Let AP be a set of propositional variables. A* Kripke structure *M over AP is a triple $M = (S, R, L)$, where*

1. *S is a finite set of states;*
2. *$R \subseteq S \times S$ is a binary relation representing state transitions;*
3. *$L : S \rightarrow 2^{AP}$ is a labeling function that assigns each state with a set of propositional variables.*

A common method of visualising a Kripke structure is to see it as a rooted graph (S, R), whose nodes are labelled by L. Here, each node is coupled with a set of propositional atoms true at that state, each node representing a state in the system. Nodes are connected through relations R and the system may transition state to state based on these connections.

[1]Here we follow the style of [66] to define ACTL syntax, the original notation for CTL is attributed to [7].

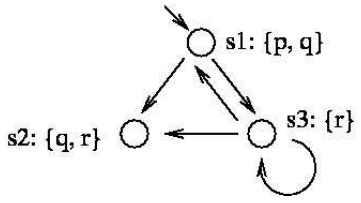


Figure 2.1: Rooted state transition graph.

We evaluate an ACTL formula over a Kripke structure. A *path* in a Kripke structure from a state is a(n) (infinite) sequence of states. Note that for a given path, the same state may occur an infinite number of times in the path (*i.e.* the path contains a loop). To simplify our following discussions, we may identify states in a path with different position subscripts, although states occurring in different positions in the path may be the same. In this way, we can say that one state precedes another in a path without much confusion. Now we can present useful notions in a formal way.

Let $M = (S, R, L)$ be a Kripke structure and $s_0 \in S$. A *path* in M starting from s_0 is denoted as $\pi = [s_0, \cdots, s_i, s_{i+1}, \cdots]$, where $(s_i, s_{i+1}) \in R$ holds for all $i \geq 0$. If $\pi = [s_0, s_1, \cdots, s_i, \cdots, s_j, \cdots]$ and $i < j$, we denote $s_i < s_j$.

Definition 2.3. *Let $M = (S, R, L)$ be a Kripke structure. Given any $s \in S$, we define whether an ACTL formula ϕ holds in M at state s. We write this as $(M, s) \models \phi$. The satisfaction relation $\models$ is defined by structural induction on ACTL formulas* [2]*:*

1. $(M,\ s) \models \top$ *and* $(M,\ s) \not\models \bot$ *for all* $s \in S$.
2. $(M,\ s) \models p$ *iff* $p \in L(s)$.
3. $(M,\ s) \models \neg p$ *iff* $(M,\ s) \not\models p$.
4. $(M,\ s) \models p \rightarrow \phi$ *iff* $(M,\ s) \not\models p$ *or* $(M,\ s) \models \phi$.
5. $(M,\ s) \models \phi_1 \wedge \phi_2$ *iff* $(M,\ s) \models \phi_1$ *and* $(M,\ s) \models \phi_2$.
6. $(M,\ s) \models \phi_1 \vee \phi_2$ *iff* $(M,\ s) \models \phi_1$ *or* $(M,\ s) \models \phi_2$.
7. $(M,\ s) \models AX\phi$ *iff* $\forall s_1$ *such that* $(s,\ s_1) \in R, (M,\ s_1) \models \phi$.
8. $(M,\ s) \models AG\phi$ *iff* $\forall \pi = [s_0,\ s_1, \ldots](s_0 = s)$ *and* $\forall s_i \in \pi, (M,\ s_i) \models \phi$.
9. $(M,\ s) \models AF\phi$ *iff* $\forall \pi = [s_0,\ s_1, \ldots](s_0 = s), \exists s_i \in \pi$ *such that* $(M,\ s_i) \models \phi$.
10. $(M,\ s) \models A[\phi_1\ U\ \phi_2]$ *iff* $\forall \pi = [s_0,\ s_1, \ldots](s_0 = s), \exists s_i \in \pi$ *such that* $(M,\ s_i) \models \phi_2$ *and* $\forall j < i, (M,\ s_j) \models \phi_1$.

[2] Following the conventions of [66].

In Clause 1, we see that a state in a model vacuously satisfies a tautology and does not satisfy a contradiction. Clause 2 states that if some propositional atom is an element of the label function mapped to the state, we can say the state satisfies the propositional atom. Similarly, Clause 3 states that if a model does not satisfy some proposition p then the same model satisfies the negation. In Clause 5 - 7, the associated truth value of some formula relies on the truth value of its subformula(s) ϕ_0 or ϕ_1 at the given state. From these clauses, the truth of AX is contingent on the truth of its subformula ϕ at the current states immediate transitory states, linked through state relations R. If any one immediate transition of s does not satisfy the subformula, AXϕ is not satisfied at (M, s). Clauses 8 - 11 also hold this property, but satisfaction of these temporal tokens is contingent on the truth values of all states available through paths π. As an example, the truth value of AFϕ relies on the reachability of some state that satisfies ϕ along each path possible from the initial state s. This means not only immediate transitions need to be checked, but all states materially connected in a path from the origin state.

From these clauses, it is clear to see that individual states satisfy propositional formulas, paths satisfy temporal properties (AX being the base case) and branching behaviour in the transition relation satisfies quantification of temporal properties. It should be noted that implication can be represented, but only where the antecedent is a propositional atom. This stems from ACTL requiring negation be restricted to propositional atoms and material implication.

Example 2.1. *In Figure 2.1 we give an example of a simple state transition graph. State s_1 is the initial state and holds the propositional atoms p and q true. Similarly s_2 holds q and r true and s_3 holds r true. Kripke structure have a two value semantics propositional atoms not true at some state are false. We unwind this transition graph to an infinite tree illustrated in Figure 2.5. From this unwound model we can see which ACTL formulas it satisfies. We can see from the initial state $M, s \models AXr$, $M, s \models AGAFr$ and $M, s \models AG(A[p\ U\ r])$. Its also the case that $M, s \not\models AFq$ and $M, s \not\models AGp$ witnessed by the counterexamples $\pi_1 = [s_1,\ s_3]$ and $\pi_2 = [s_1,\ s_2]$, respectively.*

As a convention, with respect to satisfying some Kripke structure, ACTL formulas may be referred to as a system property or specification in that when checking

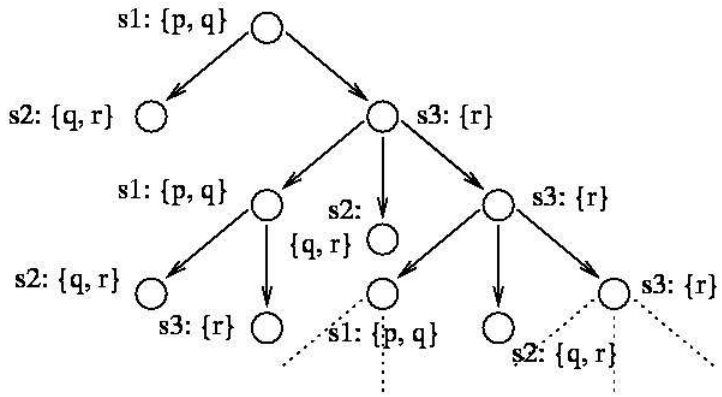


Figure 2.2: Unwound transition state diagram as an infinite tree.

the formula, we see the formula as holding some property the system should embody, or that the formula specifies some behaviour inherent in the model. Furthermore, there exists some usable overlap between formulas in the universal and existential fragments of CTL. To be able to derive counterexamples from some specification, negation needs to be restricted to propositional atoms and the resultant formula must only contain universal quantifiers over paths [27].

Now we introduce the concept of tree-like Kripke structures [27]. In tree-like local model update, Kripke structures have constraints applied to their transition graph, allowing an overall tree structure with embedded cycles or self loops.

Definition 2.4. *Let G be a directed graph. A* Strongly Connected Component *(SCC) C in G is a maximal subgraph of G such that every node in C is reachable from every other node in C along a path entirely contained within C. C is* nontrivial *iff either it has more than one node or it contains one node with a self-loop. The* component graph $c(G)$ *of G is the graph where the vertices are given by the SCCs of G, and where two vertices of $c(G)$ are connected by an edge if there exists an edge between vertices in the corresponding SCCs. Then we say a graph G is* tree-like *if*

1. *all its SCCs are cycles;*
2. *$c(G)$ is a directed tree.*

We denote a child where v, v' are two nodes in G. Then we can say that v is a child *of v' in G iff v is a child of v' in graph $c(G)$.*

We should note that condition (1) is non-trivial because some SCCs may not be cycles. For instance, in a graph $G = (V,\ E)$, where $V = \{s_1,\ s_2,\ s_3\}$ and $E = \{(s_1,\ s_2),\ (s_2,\ s_3),\ (s_3,\ s_3),\ (s_3,\ s_2)\}$, the subgraph $G' = (\{s_2,\ s_3\},\ \{(s_2,\ s_3),\ (s_3,\ s_3),\ (s_3,\ s_2)\})$ is a SCC, but it is not a cycle because edge $(s_3,\ s_3)$ also forms a self-loop.

Consider a Kripke model (M, s_0), where $M = (S, R, L)$ and $s_0 \in S$. We say that (M, s_0) is a *tree-like Kripke model* if its corresponding graph $G(M) = (S, R)$ is tree-like. In this case, we call the initial state s_0 the *root* of this tree-like model. Since a tree-like Kripke model may not be a strict tree (*e.g.* it may contain some cycles along a branch), we cannot follow the traditional notions of *child and parent* in a tree-like model. Instead, we define the following new concepts. We say state s is an *ancestor* of state s', if there is a path $\pi = [s_0, \cdots, s, \cdots, s', \cdots]$ where s' does not occur in the section $[s_0, \cdots, s]$ $s' \not< s$, $s < s'$ in π and for all $s^* \in \pi$ where $s^* < s$, $s^* \neq s'$.

s is a *parent* of s' if s is an ancestor of s' and $(s, s') \in R$. In this case, we also call s' is a *successor* of s. A state s is called *leaf* if it is not an ancestor of any other states. A tree-like model (M', s') is called a *submodel* of (M, s), if $M' = (S', R', L')$, $s' \in S'$, $S' \subseteq S$, $R' \subseteq R$, for all $s^* \in S'$, $L'(s^*) = L(s^*)$, and in M, s' is an ancestor of all other states in M'. Figure 2.3 from [27] shows an example of a tree-like model that represents a counterexample for a specific ACTL formula.

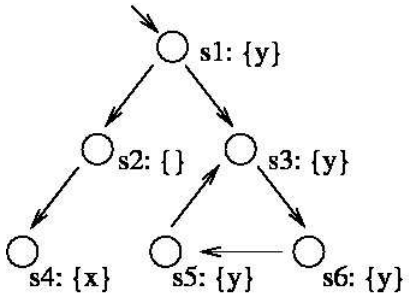


Figure 2.3: A counterexample for AG$\neg x \vee$AF$\neg y$.

Clarke et al. [27] proved an important result regarding ACTL model checking, stating that if an ACTL formula is not satisfied in a Kripke structure, then this Kripke structure must contain a tree-like counterexample with respect to this formula.

Theorem 2.1. *[27] ACTL has tree-like counterexamples.*

2.2 Tree-like Model Update

As we mentioned earlier, one major obstacle restricting the application of CTL model update in practical domains is that very often we have to deal with a large Kripke structure, and the update may also often lead to model explosion [101]. This motivates us to focus on the tree-like counterexample update. Since we can view a tree-like counterexample as a partial Kripke structure which is usually small and contains the violation of the specification, the update on the counterexample actually provides a computer aided approach for effective system modifications. Here we define what makes one update better than another proposed model modification.

2.2.1 Defining Minimal Change

Now the question becomes which criteria we are aiming for when we wish to update a tree-like ACTL Kripke model to satisfy the underlying property. Intuitively, we would like our update approach to adhere to the following general principles:

1. *Retain the original structure as much as possible;*
2. *Do not change the structure components that are irrelevant to satisfying the property;*
3. *Allow changes on both transition relations and states in the structure.*

Principles 1 and 2 are quite obvious; whenever possible, we always like to change the structure as little as possible to make it satisfy the property. Principle 3, on the other hand, means that our update should be flexible enough in order to represent rational modifications in system repairs.

Now we start to formalise the update on tree-like Kripke structures. For brevity, in this thesis we will call (M, s) a *tree-like Kripke model* without explicitly mentioning the corresponding tree-like Kripke structure $M = (S, R, L)$ where $s \in S$. We also define the difference metric $\mathsf{Diff}(X, Y) = (X \setminus Y) \cup (Y \setminus X)$, where X, Y are two sets, considering that the underlying Kripke structure models a specific system behaviours.

Definition 2.5 (Weak bisimulation)**.** *Let (M, s) and (M_1, s_1) be two tree-like Kripke models where $M = (S, R, L)$ and $M_1 = (S_1, R_1, L_1)$ and $s \in S$ and $s_1 \in S_1$. We say that a binary relation $H \subseteq S \times S_1$ is a* weak bisimulation *between (M, s) and (M_1, s_1) if:*

1. *$H(s, s_1)$;*
2. *given $v, v' \in S$ such that v is a parent of v', for all $v_1 \in S_1$ such that $H(v, v_1)$, the condition holds: (a) if v_1 is not a leaf, then there exists successor v_1' of v_1 such that $H(v', v_1')$, or (b) if v_1 is a leaf, then $H(v', v_1)$ (forth condition);*
3. *given $v_1, v_1' \in S_1$ such that v_1 is a parent of v_1', for all $v \in S$ such that $H(v, v_1)$, the condition holds: (a) if v is not a leaf, then there exists a successor v' of v such that $H(v', v_1')$, or (b) if v is a leaf, then $H(v, v_1')$ (back condition).*

Definition 2.5 is inspired from the concept of bisimulation on Kripke models in classical modal logics [8]. It is observed that for any two tree-like models, there exists at least one weak bisimulation between them. Usually, there are more than one such weak bisimulations.

Example 2.2. *In Figure 2.4 we represent two models (M, s) and (M', s'), with transition graph (S, R) and (S', R') where*

$(S, R) = (\{s_1,\ s_2,\ s_3,\ s_4,\ s_5\}, \{(s_1,\ s_2),\ (s_1,\ s_3),\ (s_3,\ s_5),\ (s_5,\ s_4),\ (s_4,\ s_3)\})$

and $(S', R') = (\{s_1',\ s_2',\ s_3'\}, \{(s_1',\ s_2'),\ (s_1',\ s_3')\})$. A weak bisimulation H can be found where $H = \{(s_1, s_1'),\ (s_2, s_2'),\ (s_5, s_3'),\ (s_4, s_3'),\ (s_3, s_3')\}$.

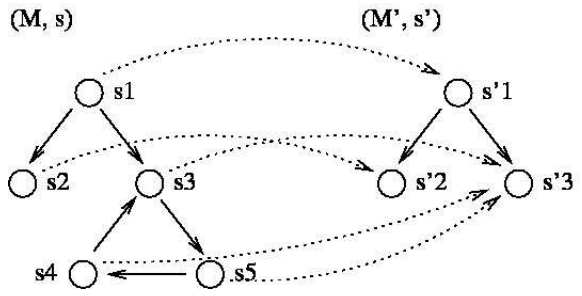


Figure 2.4: A bisimulation mapping between (M, s) and (M', s').

Definition 2.6 (Bisimulation ordering)**.** *Let* (M, s)*,* (M_1, s_1) *and* (M_2, s_2) *be three tree-like models, where* $M = (S, R, L)$*,* $M_1 = (S_1, R_1, L_1)$*,* $M_2 = (S_2, R_2, L_2)$*, and* $s \in S$*,* $s_1 \in S_1$ *and* $s_2 \in S_2$*,* H_1 *and* H_2 *be two weak bisimulations between* (M, s) *and* (M_1, s_1) *and between* (M, s) *and* (M_2, s_2) *respectively. We say that* H_1 *is* as similar as H_2*, denoted by* $H_1 \leq H_2$*, if for all nodes* $v \in S$*, the following condition holds:*

1. *there exists an ancestor* v' *of* v *such that for all* $v_1 \in S_1$ *and* $v_2 \in S_2$ *satisfying* $H_1(v', v_1)$ *and* $H_2(v', v_2)$*,* $\mathsf{Diff}(L(v'), L_1(v_1)) \subset \mathsf{Diff}(L(v'), L_2(v_2))$*; or*
2. *for all* $v_1 \in S_1$ *and* $v_2 \in S_2$ *satisfying* $H_1(v, v_1)$ *and* $H_2(v, v_2)$*,* $\mathsf{Diff}(L(v), L_1(v_1)) \subseteq \mathsf{Diff}(L(v), L_2(v_2))$*.*

We write $H_1 < H_2$ *iff* $H_1 \leq H_2$ *but* $H_2 \not\leq H_1$*.*

Definition 2.6 specifies how we compare two weak bisimulations among three tree-like models. Intuitively, if H_1 and H_2 are two weak bisimulations between (M, s) and (M_1, s_1), and between (M, s) and (M_2, s_2) respectively, then $H_1 \leq H_2$ means that M_1 represents at least the same information about M as M_2 does under H_1 and H_2 respectively.

Note that if (M_1, s_1) and (M_2, s_2) are identical, then it is still possible to have two different weak bisimulations between (M, s) and (M_1, s_1). Hence, we are always interested in that H_1 where there is no other H_1' between (M, s) and (M', s') such that $H_1' < H_1$. We call such H_1 a *minimal weak bisimulation* between (M, s) and (M', s'), which, as should be noted, is not necessarily unique.

Example 2.3. *In Figure 2.5 we have a model* (M, s) *with bisimulation relations with* (M', s') *and* (M'', s'')*. As in Example 2.2,* (M', s') *and* (M'', s'') *have the same transition graph as initially in* (S', R') *and* (M, s) *as* (S, R)*. In this example we see that* $H_1 < H_2$ *due to the difference between the labels in* s_1 *and* s_1'' *is greater than the difference between the labels in* s_1 *and* s_1' *(i.e.* $\mathsf{Diff}(L(s_1), L(s_1')) \subset \mathsf{Diff}(L(s_1), L(s_1''))$*).*

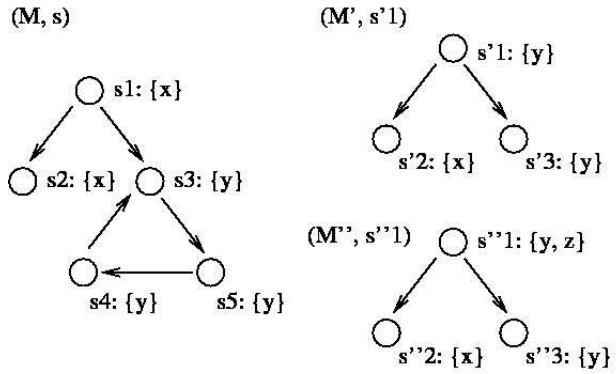


Figure 2.5: Bisimulation ordering where $H_1 < H_2$.

Definition 2.7 (Tree-like model update)**.** *Let ϕ be an ACTL formula and (M, s) a tree-like model such that $M \not\models \phi$. A tree-like model (M_1, s_1) is called a result of updating (M, s) with ϕ, denoted as $\mathsf{Update}^{\mathsf{t}}((M, s), \phi)$, if and only if*

1. *$(M_1, s_1) \models \phi$;*
2. *there is a weak bisimulation H_1 between (M, s) and (M_1, s_1) such that there does not exist another tree-like model (M_2, s_2) satisfying $(M_2, s_2) \models \phi$ and a weak bisimulation H_2 between (M, s) and (M_2, s_2) such that $H_2 < H_1$. In this case we say that (M_1, s_1) is an update result* under *H_1.*

Condition 1 in Definition 2.7 simply states that after the update, the resulting tree-like model should satisfy the updating formula. Condition 2 ensures that the resulting tree-like model is minimal from the original model under some weak bisimulation. We also use $\mathsf{Res}(\mathsf{Update}^{\mathsf{t}}((M, s), \phi))$ to denote the set of all possible resulting tree-like models of updating (M, s) with ϕ.

Example 2.4. *Consider a tree-like model M as described in Figure 2.6, which is a counterexample of $AG\neg x \vee AF\neg y$. Then according to Definition 2.7, we can verify that (M', s'_1) is a result of the update of (M, s) with $AG\neg x \vee AF\neg y$, where (M'', s''_2) is not although it also satisfies $AG\neg x \vee AF\neg y$, as (M'', s''_2) represents more changes from (M, s) than (M', s'_1) does.*

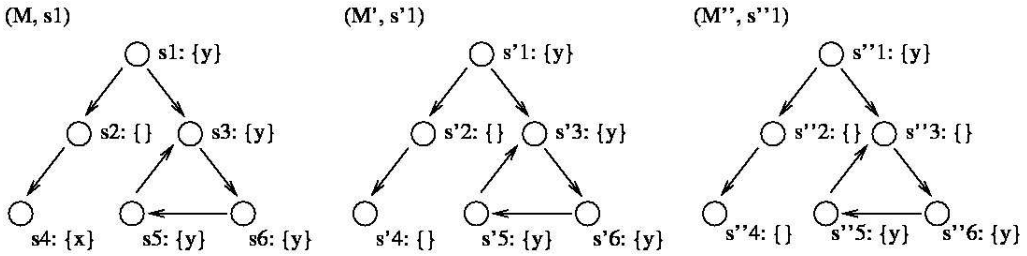


Figure 2.6: Updating (M, s) with $\mathrm{AG}\neg x \vee \mathrm{AF}\neg y$.

2.3 Semantic Properties

In this section we analyse the necessary semantic properties of tree-like local model update. We will focus our analysis on the relationship between the local model update approach and the traditional method of belief update (knowledge systems), and also review the property of satisfaction persistence that we expect local model update to satisfy.

2.3.1 Relationship to Belief Update

In [70], Katsuno and Mendelzon discovered that the original AGM revision postulates cannot precisely characterise the feature of belief update. The following alternative update postulates have been proposed. They argued that any propositional belief update operators should satisfy these postulates. For the following postulates (U1) - (U8), all occurrences of T, μ, α etc. are propositional formulas.

(U1): $T \diamond \mu \models \mu$.

(U2): *If* $T \models \mu$ *then* $T \diamond \mu \equiv T$.

(U3): *If both* T *and* μ *are satisfiable then* $T \diamond \mu$ *is also satisfiable.*

(U4): *If* $T_1 \equiv T_2$ *and* $\mu_1 \equiv \mu_2$ *then* $T \diamond \mu_1 \equiv T_2 \diamond \mu_2$.

(U5): $(T \diamond \mu) \wedge \alpha \equiv T \diamond (\mu \wedge \alpha)$.

(U6): *If* $T \diamond \mu_1 \models \mu_2$ *and* $T \diamond \mu_2 \models \mu_1$ *then* $T \diamond \mu_1 \equiv T \diamond \mu_2$.

(U7): *If* T *is complete (i.e. has a unique model) then* $(T \diamond \mu_1) \wedge (T \diamond \mu_2) \models T \diamond (\mu_1 \vee \mu_2)$.

(U8): $(T_1 \vee T_2) \diamond \mu \equiv (T_1 \diamond \mu) \vee (T_2 \diamond \mu)$.

As shown by Katsuno and Mendelzon in [70], postulates (U1) - (U8) precisely capture the minimal change criterion for update that is defined based on certain partial ordering on models.

In order to compare our tree-like model update with the traditional knowledge based update, we need to first define a tree-like update operator on ACTL formulas. Let ϕ be an ACTL formula, $M = (S, R, L)$ a tree-like Kripke structure, and $Initial(S) \subseteq S$ the *initial states* of M. For some $s \in Initial(S)$, we call (M, s) a *tree-like model* of ϕ if $(M, s) \models \phi$. We use $\mathsf{Mod^t}(\psi)$ to denote the set of all tree-like models of ψ. We also assume that the underlying language is finite, *i.e.* the set of propositional variables is finite, because Katsuno and Mendelzon's postulates (U1) - (U8) are based on finite languages [70].

Proposition 2.1. *If ψ is a satisfiable ACTL formula, then $\mathsf{Mod^t}(\psi) \neq \emptyset$.*

Given two ACTL formulas ψ and ϕ, we specify a tree-like update operator $\diamond_t$ as follows:

$$\mathsf{Mod^t}(\psi \diamond_\mathsf{t} \phi) = \bigcup_{(M,s)\in\mathsf{Mod^t}(\psi)} \mathsf{Res}(\mathsf{Update^t}((M, s), \phi)). \tag{2.1}$$

Theorem 2.2. *Operator $\diamond_t$ satisfies all Katsuno and Mendelzon update postulates (U1) - (U8), in the sense that for each form $\psi \models \phi$ in the postulates, we replace it as $\mathsf{Mod^t}(\psi) \subseteq \mathsf{Mod^t}(\phi)$.*

Proof. Here we show that $\diamond_\mathsf{t}$ satisfies (U1) through (U8).

To prove $\diamond_\mathsf{t}$ satisfies (U1) we show $\mathsf{Mod^t}(T \diamond \mu) \subseteq \mathsf{Mod^t}(\mu)$. Since $\mathsf{Mod^t}(T \diamond_\mathsf{t} \mu) = \bigcup_{(M,s)\in\mathsf{Mod^t}(T)} \mathsf{Res}(\mathsf{Update^t}((M, s), \mu))$ where for each $(M, s) \in \mathsf{Mod^t}(T)$, $\mathsf{Update^t}((M, s), \mu) \in \mathsf{Mod^t}(\mu)$ so $\mathsf{Mod^t}(T \diamond \mu) \subseteq \mathsf{Mod^t}(\mu)$.

(U2) In proving if $T \models \mu$ then $T \diamond \mu \equiv T$, this equates to $\mathsf{Mod^t}(T) \subseteq \mathsf{Mod^t}(\mu)$ then $\mathsf{Mod^t}(T \diamond_\mathsf{t} \mu) \subseteq \mathsf{Mod^t}(T)$. From the prior definition we know $\mathsf{Mod^t}(T \diamond_\mathsf{t} \mu) = \bigcup_{(M,s)\in\mathsf{Mod^t}(T)} \mathsf{Res}(\mathsf{Update^t}((M, s), \mu))$. As $T \models \mu$, every model selected for μ will be in $\mathsf{Mod^t}(T)$, such $\mathsf{Update^t}((M, s), \mu) = (M, s)$ and $T \diamond_\mathsf{t} \mu \equiv T$ when $T \models \mu$.

(U3) By definition for any formula μ that is satisfiable, $\mathsf{Mod^t}(\mu) \neq \emptyset$, *i.e.* there will be a set of tree-like models associated with any satisfiable formula, for $T \diamond \mu$ to be satisfiable it will have corresponding tree-like models. Assume $T \diamond \mu$ is not satisfiable, but T and μ are. This is equivalent to saying $\mathsf{Mod^t}(T) \neq \emptyset$, $\mathsf{Mod^t}(\mu) \neq \emptyset$ and $\mathsf{Mod^t}(T \diamond_\mathsf{t} \mu) = \emptyset$. We update T by μ, $\mathsf{Res}(\mathsf{Update^t}((M, s), \mu))$ will have tree-like models as μ is satisfiable, and since $\mathsf{Mod^t}(T) \neq \emptyset$ there will be some $(M, s) \in$

$\mathsf{Mod}^{\mathsf{t}}(T)$ selected where $(M,s) \in \mathsf{Res}(\mathsf{Update}^{\mathsf{t}}((M,s),\mu))$, thus $\mathsf{Mod}^{\mathsf{t}}(T \diamond_{\mathsf{t}} \mu) \neq \emptyset$. Thus the update operator $\diamond_t$ produces satisfiable formulas given satisfiable formulas are used.

Proving $\diamond_{\mathsf{t}}$ for (U4) it is sufficient to prove that for all $(M,s) \in \mathsf{Mod}^{\mathsf{t}}(T_1)$ that if $\mathsf{Mod}^{\mathsf{t}}(T_1) = \mathsf{Mod}^{\mathsf{t}}(T_2)$ and $\mathsf{Mod}^{\mathsf{t}}(\mu_1) = \mathsf{Mod}^{\mathsf{t}}(\mu_2)$ then $\mathsf{Res}(\mathsf{Update}^{\mathsf{t}}((M,s),\mu_1)) = \mathsf{Res}(\mathsf{Update}^{\mathsf{t}}((M,s),\mu_2))$. To show this we prove that $\mathsf{Res}(\mathsf{Update}^{\mathsf{t}}((M,s),\mu_1)) \subseteq \mathsf{Res}(\mathsf{Update}^{\mathsf{t}}((M,s),\mu_2))$ and its back condition $\mathsf{Res}(\mathsf{Update}^{\mathsf{t}}((M,s),\mu_2)) \subseteq \mathsf{Res}(\mathsf{Update}^{\mathsf{t}}((M,s),\mu_1))$. Assume $\mathsf{Mod}^{\mathsf{t}}(T_1) = \mathsf{Mod}^{\mathsf{t}}(T_2)$ and $\mathsf{Mod}^{\mathsf{t}}(\mu_1) = \mathsf{Mod}^{\mathsf{t}}(\mu_2)$ but $\mathsf{Res}(\mathsf{Update}^{\mathsf{t}}((M,s),\mu_1)) \not\subseteq \mathsf{Res}(\mathsf{Update}^{\mathsf{t}}((M,s),\mu_2))$. Let the model $(M',s') \in \mathsf{Res}(\mathsf{Update}^{\mathsf{t}}((M,s),\mu_1))$ and $(M',s') \in \mathsf{Mod}^{\mathsf{t}}(T_1)$. Suppose $(M',s') \notin \mathsf{Res}(\mathsf{Update}^{\mathsf{t}}((M,s),\mu_2))$. Then $(M'',s'') \in \mathsf{Res}(\mathsf{Update}^{\mathsf{t}}((M,s),\mu_2))$ such $H'' < H'$ and $(M'',s'') \in \mathsf{Mod}^{\mathsf{t}}(T_1)$. This contradicts $(M',s') \in \mathsf{Res}(\mathsf{Update}^{\mathsf{t}}((M,s),\mu_1))$ and $\mathsf{Res}(\mathsf{Update}^{\mathsf{t}}((M,s),\mu_1)) \subseteq \mathsf{Res}(\mathsf{Update}^{\mathsf{t}}((M,s),\mu_2))$.

Now we prove $\mathsf{Res}(\mathsf{Update}^{\mathsf{t}}((M,s),\mu_2)) \subseteq \mathsf{Res}(\mathsf{Update}^{\mathsf{t}}((M,s),\mu_1))$. Assume $\mathsf{Mod}^{\mathsf{t}}(T_1) = \mathsf{Mod}^{\mathsf{t}}(T_2)$ and $\mathsf{Mod}^{\mathsf{t}}(\mu_1) = \mathsf{Mod}^{\mathsf{t}}(\mu_2)$ but $\mathsf{Res}(\mathsf{Update}^{\mathsf{t}}((M,s),\mu_2)) \not\subseteq \mathsf{Res}(\mathsf{Update}^{\mathsf{t}}((M,s),\mu_1))$. Let the model $(M',s') \in \mathsf{Res}(\mathsf{Update}^{\mathsf{t}}((M,s),\mu_2))$ and $(M',s') \in \mathsf{Mod}^{\mathsf{t}}(T_2)$. Suppose $(M',s') \notin \mathsf{Res}(\mathsf{Update}^{\mathsf{t}}((M,s),\mu_1))$. Then $(M'',s'') \in \mathsf{Res}(\mathsf{Update}^{\mathsf{t}}((M,s),\mu_1))$ such $H'' < H'$ and $(M'',s'') \in \mathsf{Mod}^{\mathsf{t}}(T_2)$. This contradicts $(M',s') \in \mathsf{Res}(\mathsf{Update}^{\mathsf{t}}((M,s),\mu_2))$ and $\mathsf{Res}(\mathsf{Update}^{\mathsf{t}}((M,s),\mu_2)) \subseteq \mathsf{Res}(\mathsf{Update}^{\mathsf{t}}((M,s),\mu_1))$.

To prove that $\diamond_{\mathsf{t}}$ satisfies (U5), it is sufficient to show that for each $(M,s) \in \mathsf{Mod}^{\mathsf{t}}(T)$, $\mathsf{Res}(\mathsf{Update}^{\mathsf{t}}((M,s),\mu)) \cap \mathsf{Mod}^{\mathsf{t}}(\alpha) \subseteq \mathsf{Res}(\mathsf{Update}^{\mathsf{t}}((M,s),\mu \wedge \alpha))$. Consider a tree-like model $(M',s') \in \mathsf{Res}(\mathsf{Update}^{\mathsf{t}}((M,s),\mu)) \cap \mathsf{Mod}^{\mathsf{t}}(\alpha)$. Suppose $(M',s') \notin \mathsf{Res}(\mathsf{Update}^{\mathsf{t}}((M,s),\mu \wedge \alpha))$. Then this implies two cases: (a) $(M',s') \not\models \mu \wedge \alpha$; (b) there exists another tree-like model $(M'',s'') \in \mathsf{Mod}^{\mathsf{t}}(\mu \wedge \alpha)$ such that $H'' < H'$, where H', H'' are weak bisimulations between (M,s) and (M',s'), (M,s) and (M'',s'') respectfully. If it is the case (a) then $(M',s') \notin \mathsf{Res}(\mathsf{Update}^{\mathsf{t}}((M,s),\mu)) \cap \mathsf{Mod}^{\mathsf{t}}(\alpha)$, so the result holds. If (b) is the case, then it means that $(M',s') \notin \mathsf{Res}(\mathsf{Update}^{\mathsf{t}}((M,s),\mu))$ according to Definition 2.7, and hence $(M',s') \notin \mathsf{Res}(\mathsf{Update}^{\mathsf{t}}((M,s),\mu)) \cap \mathsf{Mod}^{\mathsf{t}}(\alpha)$. The result also holds.

For $\diamond_{\mathsf{t}}$ to satisfy (U6) it is sufficient to prove for any given $(M,s) \in \mathsf{Mod}^{\mathsf{t}}(T)$ if $\mathsf{Res}(\mathsf{Update}^{\mathsf{t}}((M,s),\mu_1)) \subseteq \mathsf{Mod}^{\mathsf{t}}(\mu_2)$ and $\mathsf{Res}(\mathsf{Update}^{\mathsf{t}}((M,s),\mu_2)) \subseteq \mathsf{Mod}^{\mathsf{t}}(\mu_1)$ then $\mathsf{Res}(\mathsf{Update}^{\mathsf{t}}((M,s),\mu_1)) = \mathsf{Res}(\mathsf{Update}^{\mathsf{t}}((M,s),\mu_2))$. To show this we prove that $\mathsf{Res}(\mathsf{Update}^{\mathsf{t}}((M,s),\mu_1)) \subseteq \mathsf{Res}(\mathsf{Update}^{\mathsf{t}}((M,s),\mu_2))$. Let an updated tree-like model $(M',s') \in \mathsf{Res}(\mathsf{Update}^{\mathsf{t}}((M,s),\mu_1))$. Then $(M',s') \models \mu_2$. Suppose $(M',s') \notin \mathsf{Res}(\mathsf{Update}^{\mathsf{t}}((M,s),\mu_2))$, then there exists a different admissible model $(M'',s'') \in \mathsf{Res}(\mathsf{Update}^{\mathsf{t}}((M,s),\mu_2))$ such that there is a bisimulation ordering where $H'' < H'$. Also note that $(M'',s'') \models \mu_1$. This contradicts $(M',s') \in \mathsf{Res}(\mathsf{Update}^{\mathsf{t}}((M,s),\mu_1))$. With this we have $\mathsf{Res}(\mathsf{Update}^{\mathsf{t}}((M,s),\mu_1)) \subseteq \mathsf{Res}(\mathsf{Update}^{\mathsf{t}}((M,s),\mu_2))$.

Now we prove that $\mathsf{Res}(\mathsf{Update}^{\mathsf{t}}((M,s),\mu_2)) \subseteq \mathsf{Res}(\mathsf{Update}^{\mathsf{t}}((M,s),\mu_1))$. Let an updated tree-like model $(M',s') \in \mathsf{Res}(\mathsf{Update}^{\mathsf{t}}((M,s),\mu_2))$. Then $(M',s') \models \mu_1$. Suppose $(M',s') \notin \mathsf{Res}(\mathsf{Update}^{\mathsf{t}}((M,s),\mu_1))$, then there exists a different admissible model $(M'',s'') \in \mathsf{Res}(\mathsf{Update}^{\mathsf{t}}((M,s),\mu_1))$ such that there is a bisimulation ordering where $H'' < H'$. Also note that $(M'',s'') \models \mu_2$. This contradicts $(M',s') \in \mathsf{Res}(\mathsf{Update}^{\mathsf{t}}((M,s),\mu_2))$. With this we have $\mathsf{Res}(\mathsf{Update}^{\mathsf{t}}((M,s),\mu_2)) \subseteq \mathsf{Res}(\mathsf{Update}^{\mathsf{t}}((M,s),\mu_1))$.

(U7) If T is complete (*i.e.* has a unique model) then $(T \diamond \mu_1) \wedge (T \diamond \mu_2) \models T \diamond (\mu_1 \vee \mu_2)$. We need to prove $\mathsf{Res}(\mathsf{Update}^{\mathsf{t}}((M,s),\mu_1)) \cap \mathsf{Res}(\mathsf{Update}^{\mathsf{t}}((M,s),\mu_2)) \subseteq \mathsf{Res}(\mathsf{Update}^{\mathsf{t}}((M,s),\mu_1 \vee \mu_2))$, where (M,s) is the unique model of T (noting T is complete). Let $(M',s') \in \mathsf{Res}(\mathsf{Update}^{\mathsf{t}}((M,s),\mu_1)) \cap \mathsf{Res}(\mathsf{Update}^{\mathsf{t}}((M,s),\mu_2))$. Suppose $(M',s') \notin \mathsf{Res}(\mathsf{Update}^{\mathsf{t}}((M,s),\mu_1 \vee \mu_2))$. Then there exists an admissible model $(M'',s'') \in \mathsf{Res}(\mathsf{Update}^{\mathsf{t}}((M,s),\mu_1 \vee \mu_2))$ with a bisimulation relation where $H'' < H'$. Note that $(M'',s'') \models \mu_1 \vee \mu_2$. If $(M'',s'') \models \mu_1$, then it implies $(M',s') \notin \mathsf{Res}(\mathsf{Update}^{\mathsf{t}}((M,s),\mu_1))$. If $(M'',s'') \models \mu_2$ then it implies $(M',s') \notin \mathsf{Res}(\mathsf{Update}^{\mathsf{t}}((M,s),\mu_2))$. In both cases, we have $(M'',s'') \notin \mathsf{Res}(\mathsf{Update}^{\mathsf{t}}((M,s),\mu_1)) \cap \mathsf{Res}(\mathsf{Update}^{\mathsf{t}}((M,s),\mu_2))$. This proves the result.

Now we prove that $\diamond_{\mathsf{t}}$ satisfies (U8). From (1), it is clear that: $\mathsf{Mod}^{\mathsf{t}}((T_1 \vee T_2) \diamond_{\mathsf{t}} \mu) = \bigcup_{(M,s)\in\mathsf{Mod}^{\mathsf{t}}(T_1 \vee T_2)} \mathsf{Res}(\mathsf{Update}^{\mathsf{t}}((M,s),\mu)) = \bigcup_{(M,s)\in\mathsf{Mod}^{\mathsf{t}}(T_1)} \mathsf{Res}(\mathsf{Update}^{\mathsf{t}}((M,s),\mu)) \cup \bigcup_{(M,s)\in\mathsf{Mod}^{\mathsf{t}}(T_2)} \mathsf{Res}(\mathsf{Update}^{\mathsf{t}}((M,s),\mu)) = \mathsf{Mod}^{\mathsf{t}}(T_1 \diamond_{\mathsf{t}} \mu) \cup \mathsf{Mod}^{\mathsf{t}}(T_2 \diamond_{\mathsf{t}} \mu)$. This means $\diamond_{\mathsf{t}}$ satisfies postulate (U8). □

2.3.2 Persistence Properties

An essential semantic property we should study in relation to model update is so called *persistence*. That is, for a given tree-like model (M, s) and two ACTL formulas ϕ and ψ, where $(M, s) \not\models \phi$ and $(M, s) \models \psi$, after updating (M, s) with ϕ we obtain a new tree-like local model (M', s') such that $(M', s') \models \phi$. Then we would like to know whether ψ still holds in the new model: $(M', s') \models \psi$. In general updating a model with one formula may affect the satisfaction of other formulas in the resulting model. Study on the persistence property in a local model update is important, because this will provide essential information of how a specific local model update influences other properties that the system originally obeys. The following general result indicates that our update does not affect those irrelevant formulas' satisfactions in the resulting local model.

To this aim, we first introduce a useful notion. For a given ACTL formula ϕ, we use $Var(\phi)$ to denote the set of all propositional variables (atoms) occurring in ϕ. The following result is obvious.

Proposition 2.2. *Let (M, s) be a tree-like model, ϕ and ψ two ACTL formulas such that $Var(\phi) \cap Var(\psi) = \emptyset$, and (M', s') a tree-like model resulting from the update of (M, s) with ϕ. Then $(M', s') \models \psi$ iff $(M, s) \models \phi$.*

However, the situation becomes complicated when ϕ and ψ share some common propositional variables. In general, the persistence property does not hold any more in a local model update when two formulas share common propositional variables. What makes this problem meaningful and challenging is to identify some useful cases for which the local model update preserves the persistence property for a class of ACTL formulas.

Definition 2.8 (Strict extension)**.** *A tree-like model (M, s) is called a strict extension of (M', s') if, for each path in (M', s') $\pi' = [s_0, s_1, \ldots](s_0 = s')$, there is at most one path in (M, s) $\pi = [s_0, \ldots, s_k, s_{k+1}, \ldots](s_0 = s')$, such that for all $s_i \leq s_k,\ s_i \in \pi'$, and for all $s_k < s_j,\ s_j \notin \pi'$.*

Theorem 2.2 indicates that during a tree-like model update, our approach will not affect those irrelevant formulas' satisfactions in the resulting model.

Intuitively, if (M, s) is a strict extension of (M', s'), then (M, s) does not contain any more branches than (M', s') does. Strict extensions represent some interesting cases in tree-like local model updates. Quite often, an update result may be obtained by only *cutting off* or *extending* some paths of the original local model. The following theorem reveals an important persistence property associated to strict extensions.

Theorem 2.3. *Let (M, s) be a tree-like model and ϕ an ACTL formula. Then the following results hold:*

1. *If (M', s') is a result of updating (M, s) with ϕ, and it is a strict extension of (M, s), then for any ACTL formula ψ not containing operator AG, $(M, s) \models \psi$ implies $(M', s') \models \psi$;*
2. *If (M', s') is a result of updating (M, s) with ϕ, and (M, s) is a strict extension of (M', s'), then for any ACTL formula ψ only containing operator AG, $(M, s) \models \psi$ implies $(M', s') \models \psi$;*

Proof. We prove Result 1 by induction on the structure of formula ψ. More specifically, it is sufficient to prove the cases of propositional formula ψ, AXψ, AFψ, and A[ϕ U ψ]. Let $M = (S, R, L)$ and $M' = (S', R', L')$. From Definition 2.8, we know that (M', s') must contain the same branches as (M, s) contains, and $s = s'$. We first consider that ϕ is just a propositional formula. Then we have $(M, s) \models \psi$ iff $L(s) \models \phi$. This means $(M', s') = (M', s) \models \psi$.

Now suppose ϕ is of the form AXψ. Since $(M, s) \models$ AXψ we know that for each $s^* \in S$ such that $(s, s^*) \in R$, $(M, s^*) \models \psi$. Again, because (M', s') is a strict extension of (M, s), all such (s, s^*) are also in R', so $(M', s^*) \models \psi$. Furthermore, there is no other new state $s^\dagger$ such that $(s, s^\dagger) \notin R$. So $(M', s') \models$ AXψ as well.

Suppose ϕ is of the form AFψ. From $(M, s) \models$ AFψ, we know that for each path $\pi = [s, \ldots]$ in M, there exists some $s_k \in \pi$ such that $(M, s_k) \models \psi$. This path must be in M', so we have $(M', s_k) \models \psi$. On the other hand, (M', s') is a strict extension of (M, s), in M' there does not exist a path of the form $\pi' = [s', \ldots, s_k, s_{k+1}, \ldots]$ where states $s_k \in S$, and $s_{k+1}, \ldots \in S'$ and another path $\pi'' = [s', \ldots, s_k, s'_{k+1}, \ldots]$

is in M and also in $M'(s_{k+1} \neq s'_{k+1})$. That means, it is not possible that state s_k leads to two different paths in M'. This implies $(M', s) \models \mathrm{AF}\psi$.

Suppose ϕ is of the form $\mathrm{A}[\psi_1 \ \mathrm{U} \ \psi_2]$. From $(M, s) \models \mathrm{A}[\psi_1 \ \mathrm{U} \ \psi_2]$, we know that for each path $\pi = [s, \ldots]$ there exists some $s_k \in \pi$ such that $(M, s_k) \models \psi_2$ and for all $j < k, (M, s_j) \models \psi_1$. This path must also be in M', so we have $(M', s_k) \models \psi_2$ and $\forall j < k, (M', s_j) \models \psi_1$. On the other hand, (M', s') is a strict extension of (M, s), in M' there does not exist a path of the form of $\pi' = [s', \ldots, s_k, s_{k+1}]$ where states $s_k \in S$ and $s_{k+1}, \ldots \in S'$ and another path $\pi'' = [s', \ldots, s_k, s'_{k+1}, \ldots]$ is in M and also in $M'(s_{k+1} = s'_{k+1})$. The state s_k which terminates the until clause and all $j < k$ holding will not lead to a different path in M'. This implies $(M', s) \models \mathrm{A}[\psi_1 \ \mathrm{U} \ \psi_2]$.

Result 2 can be proved similarly. It is sufficient to prove the case for the formulas only containing AG. Let $M = (S, R, L)$ and $M' = (S', R', L')$ where (M, s) is a strict extension of (M', s'). From Definition 2.8 we know (M, s) must contain the same branches as (M', s') contains and $s = s'$. For formulas containing AG, if some path in $(M, s)\pi = [s, \ldots, s_k, s_{k+1}, \ldots]$ where $(M, s) \models \phi$ there will be a corresponding $\pi' = [s, \ldots, s_k]$ in (M', s'), where $(M', s') \models \phi$, and π extends π', by definition of AG if all states in (M', s') satisfy ψ, $(M', s') \models \mathrm{AG}\psi$.

□

Example 2.5. *The tree-like local model (M, s) depicted in the following figure represents a counterexample of ACTL formula $A[a \ U \ b] \vee AX(a)$. From Definition 2.7, it can be checked that model (M', s) on the right side in Figure 2.7 is one possible result of updating (M, s) with formula $A[a \ U \ (\neg a \wedge b)] \vee AX(a)$. Clearly, (M', s) is a strict extension of (M, s) in this case. Consider another formula $AF(AX(a \vee c))$. It is observed that $(M, s) \models AF(AX(a \vee c))$. According to Result 1 in Theorem 2.3, we should have $(M', s) \models AF(AX(a \vee c))$ as well.*

□

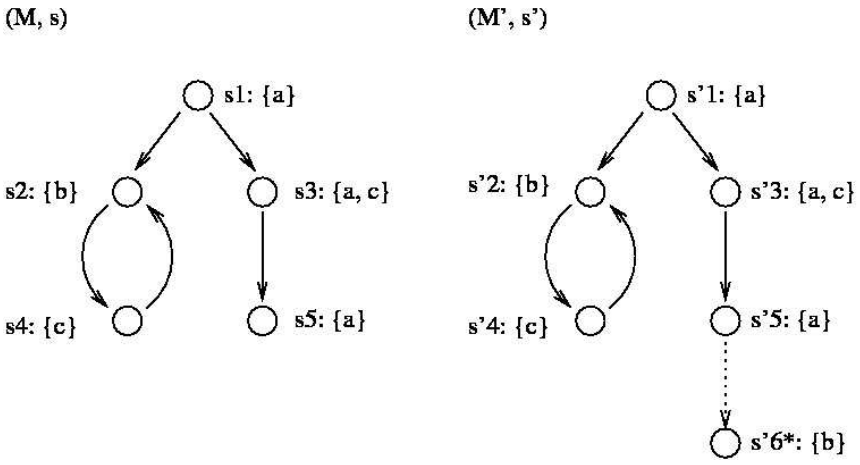


Figure 2.7: Updating (M, s) with A[a U $(\neg a \wedge b)$]$\vee$ AX(a) preserves the persistence of AF(AX($a \vee c$)).

2.4 Computational Properties

In this section we study the computational properties in relation to tree-like local model update. We will first provide the main complexity result of local model update, and then study the computational issues of some typical tree-like local model updates.

2.4.1 Complexity Results

We first provide the main complexity result of tree-like local model update as follows.

Theorem 2.4. *Let (M, s) and (M', s') be two tree-like models, H a weak bisimulation between (M, s) and (M', s'), and ϕ an ACTL formula. Deciding whether (M', s') is a result of updating (M, s) with ϕ under H as defined in Definition 2.7 is co-NP-complete.*

Proof: Membership. To show the membership, we have first proved the following result: Given models (M, s), (M', s') and a weak bisimulation H between (M, s) and (M', s'). There exists a model (M'', s'') and a weak bisimulation H'' between (M, s) and (M'', s'') such that $H'' < H$, if there exists a model $(M^\dagger, s^\dagger)$, whose size is in the polynomial size of (M, s), and (M', s'), and a $H^\dagger$ between (M, s) and $(M^\dagger, s^\dagger)$ such that $H^\dagger < H$.

Suppose $M' = (S', R', L')$ and $s' \in S'$. It is well known that checking whether $(M', s') \models \phi$ can be done in $\mathcal{O}(|\phi| \cdot (|S'| + |R'|))$. Now we need to check the minimality of (M', s') according to Definition 2.5 and 2.6. For doing that, we consider the complement of the problem: checking whether (M', s') is not an update result. First, we guess another tree-like model (M'', s'') and a weak bisimulation H'' between (M, s) and (M', s'). From the above result, we know that it is sufficient to just guess a (M'', s'') which is in polynomial size of (M, s) and (M', s'). According to Definition 2.5, on the other hand, guessing a H'' between (M, s) and (M'', s'') can also be done in polynomial time. Then we check whether $(M'', s'') \models \phi$. Obviously, checking whether $(M'', s'') \models \phi$ is in polynomial time. From 2.6, it is easy to see that deciding whether $H'' < H$ is also in polynomial time. So deciding whether (M', s') is not a result is in NP. That is, the original problem is in co-NP.

(Hardness). It is well known that the validity problem for a propositional formula is co-NP-complete. Given a propositional formula ϕ, we construct a transformation from the problem of deciding the validity of ϕ to a tree-like model update in polynomial time. Let X be the set of all variables occurring in ϕ, and a, b two new variables which do not occur in X. We denote $\neg X = \bigwedge_{x_i \in X} \neg x_i$. Then, we specify a tree-like Kripke model based on the variable set

$$X \cup \{a, b\} : M = (\{s_0, s_1\}, \{(s_0, s_1), (s_1, s_1)\}, L), \text{ where } L(s_0) = \emptyset \text{ and } L(s_1) = X.$$

Now we define a new formula $\mu = \mathrm{AX}(((\phi \to a) \wedge (\neg X \wedge b)) \vee (\neg\phi \wedge a))$. Clearly formula $((\phi \to a) \wedge (\neg X \wedge b)) \vee (\neg\phi \wedge a)$ is satisfiable and $s_1 \not\models ((\phi \to a) \wedge (\neg X \wedge b)) \vee (\neg\phi \wedge a)$. So $(M, s_0) \not\models \mu$. Now we consider the update of (M, s_0) with μ. We define a new tree-like model $M' = (\{s_0', s_1'\}, \{(s_0', s_1'), (s_1', s_1')\}, L')$, where $L'(s_0') = L(s_0)$ and $L'(s_1') = \{a, b\}$. The weak bisimulation H between (M, s_0) and (M', s_0') is defined as $H(s_0, s_0')$ and $H(s_1, s_1')$. Then it is not difficult to verify that (M', s_0') is a result of updating (M, s_0) with μ iff ϕ is valid. This completes our proof. □

Theorem 2.4 provides an essential computational insight for tree-like local model update. It implies that, unless P=NP, it is unlikely to develop a polynomial time algorithm to compute an update result. Indeed, our implementation algorithm for local model update runs in exponential time generally. Furthermore, without giving

the specific bisimulation in the input, the model checking complexity as stated in Theorem 2.4 will be in Π_2^P.

2.4.2 Computing Typical Tree-Like Local Model Updates

Although computing local model update is generally expensive, we have observed that many updates with typically ACTL formulas can actually be achieved in a more effective way. In the following, we provide complexity results for these typical updates.

Theorem 2.5. *Let (M, s) be a tree-like Kripke model, where $M = (S, R, L)$ and $s \in S$, ϕ and ψ two propositional formulas. Then the following results hold.*

1. *If $(M, s) \not\models AX\phi$, then a resulting tree-like model (M', s') can be computed in time $\mathcal{O}(|R| \cdot 2^{|var(\phi)|})$;*
2. *If $(M, s) \not\models AG\phi$, then a resulting tree-like model (M', s') can be computed in time $\mathcal{O}(|S| \cdot 2^{|var(\phi)|})$;*
3. *If $(M, s) \not\models AF\phi$, then a resulting tree-like model (M', s') can be computed in time $\mathcal{O}(|R| \cdot |S| \cdot 2^{|var(\phi)|})$;*
4. *If $(M, s) \not\models A[\phi U \psi]$, then a resulting tree-like model (M', s') can be computed in time $\mathcal{O}(|R| \cdot |S| \cdot 2^{|var(\phi)|})$;*
5. *If $(M, s) \not\models AG(\phi \rightarrow AF(\psi))$, then a resulting tree-like model (M', s') can be computed in time $\mathcal{O}(|R| \cdot |S| \cdot 2^{|var(\phi)|})$.*

Proof. 1. If (M, s) is a counterexample of $\mathrm{AX}(\phi)$ then there exists some $s_1 \in S$ such $(s, s_1) \in R$, $L(s_1) \not\models \phi$, or there is no $s_1 \in S$ such $(s, s_1) \in R$ and thus $(M, s) \not\models \phi$.

Case 1. There exists $(s, s_1) \in R$. There are transitions from the state s and there are two possibilities for update. Firstly, there is no transition from the state which

satisfies the property, *i.e.* all s_1 such that $(s, s_1) \in R, L(s_1) \not\models \phi$. Semantics of AX require that at least one state exists transitioning from the origin. We can perform one of two updates to each state s_1, on the condition that one of the states maintains $(s, s_1) \in R$ and $L(s_1) \models \phi$. In this case, we can perform one of two updates to each state s_1, on the condition that one of the states maintains $(s, s_1) \in R$ and $L(s_1) \models \phi$. Possibilities include removing a transition, effecting $R' = R - (s, s_1)$ or modifying the labelling function of s_1 to s'_1, such that under labelling function $L', L'(s'_1) \models \phi$ and $\mathsf{Diff}(L(s), L'_1(s'_1))$ is minimal.

We remove from possibility the update which removes all transitions from the selected state as a method for satisfying the formula. An update (M', s') can be derived from (M, s) by updating some s_1, through the labelling and all other outward transitions being removed. All possible updates can be derived using this technique, by applying combinations of techniques while assuring one transition which satisfies the condition AXϕ exists.

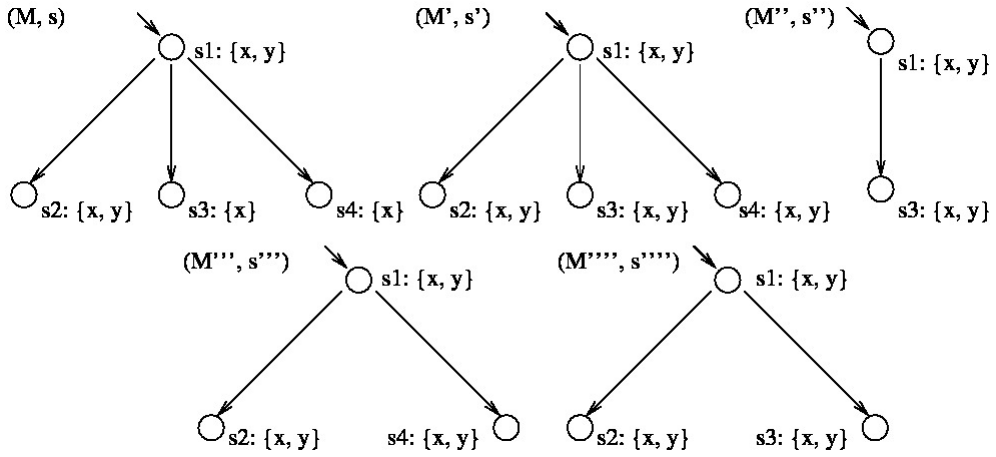


Figure 2.8: Possibilities for update for (M, s) by property AX(y).

Case 2. If there exists $s_1 \in S$, where $(s, s_1) \in R$, and $s_1 \models \phi$ but $s \not\models$ AXϕ, apply the prior case, but a state which satisfies the sub-formula is no longer required, this allows the update case where all unsatisfied branches are removed.

Case 3. There is no $s_1 \in S$, such $(s, s_1) \in R$. We extend s by s^*, such $S = S \cup \{s^*\}, R = R \cup \{(s, s^*)\}$ and $L(s^*) = L(s)$, but $L(s^*) \models \phi$ and $\mathsf{Diff}(L(s), L'(s^*))$ is minimal.

Now consider complexity for AXϕ. For each branch from the original state, we need to update exactly one state, costing time $\mathcal{O}(2^{|var(\phi)|})$[3]. Since we need to span over each relation in R from the origin, time complexity becomes $\mathcal{O}(|R| \cdot 2^{|var(\phi)|})$.

2. In a counterexample (M, s) of AGϕ, there must exist one or more paths in $(M, s)\pi = [s_0, s_1, \ldots](s_0 = s)$ such that for some $s \in \pi, L(s_1) \not\models \phi$. In this case every state along every path is required to satisfy the formula. To satisfy this formula, two updates can be applied which can affect satisfaction. One method is modification of the label function of s_1 to s_1' such that for $L', L'(s_1') \models \phi$ and $\mathsf{Diff}(L(s_1), L'(s_1'))$ is minimal. Another tree-like structure preserving update is branch pruning, as described in persistence properties, where $R' = R - (s_{i-1}, s_i)$, where $s_i \not\models \phi$.

With these two methods an update (M', s') can be derived from (M, s) by updating some s_1, though through combinations of the previous techniques such that there is no $s_1 \not\models \phi$.

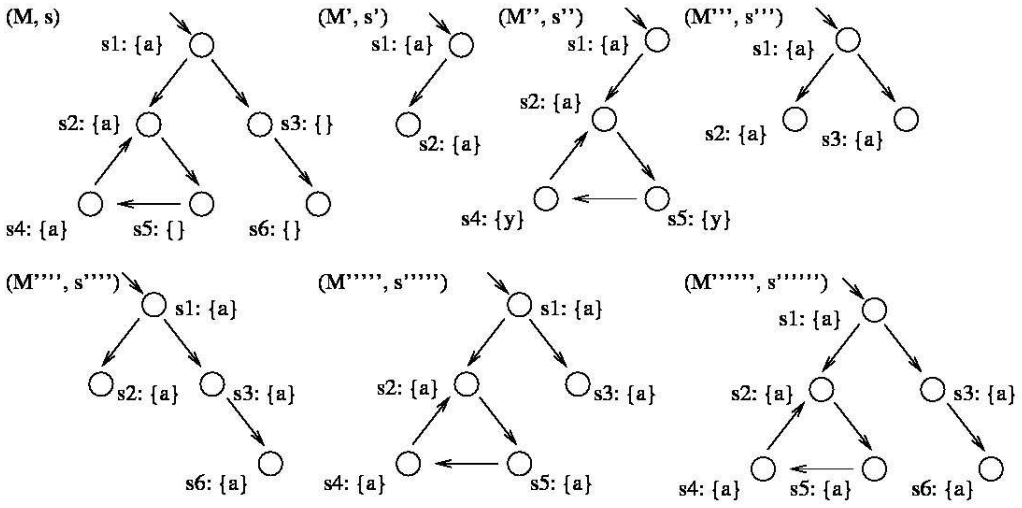


Figure 2.9: Possibilities for update for (M, s) by property AG(a).

Consider the complexity for this procedure. For each state in paths from the origin which do not satisfy the given propositional subformula we need to update, which costs time $\mathcal{O}(2^{|var(\phi)|})$. As this needs to be performed over each unsatisfied state, time complexity becomes $\mathcal{O}(|S| \cdot 2^{|var(\phi)|})$.

[3]Such complexity $\mathcal{O}(2^{|var(\phi)|})$ for individual state minimal update is inevitably inherited from the classical model based update approach [99].

3. Since (M, s) is a counterexample of AFϕ then there must be a path in $(M, s)\pi = [s_0, s_1, \ldots](s_0 = s)$ such that for each $s \in \pi, L(s_i) \not\models \phi$. There are two cases.

Case 1. π is a finite path. Recall that under weak bisimulation minimal change principle, whenever possible, we try to update the tree-like model at as lowest level as possible. In this case, we simply update the last state s_k in π to s'_k, such that under the new model's labelling function L', $L'(s'_k) \models \phi$ and $\mathsf{Diff}(L(s_k), L'(s'_k))$ is minimal. If there is no other path in (M, s) which violates Fϕ, then it is easy to see the new model (M', s') obtained in this way is a resultant model with respect to Definition 2.7.

Case 2. π is an infinite path, that is π ends up with a loop:
$\pi = [s_0,\ s_1,\ \ldots,\ s_k,\ s_{k+1},\ \ldots,\ s_{k+h},\ s_k]$, as shown in Figure 2.10.

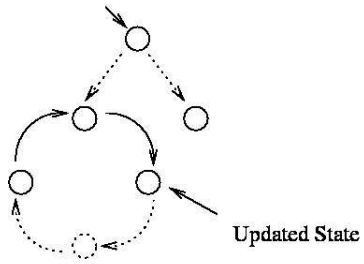


Figure 2.10: A case of update with AFϕ.

In this case, the lowest level of path π is the state s_{k+h}: the last state in the loop back to the loop entry state s_k (*i.e.* the arrow pointed state in 2.10). We update s_{k+h} to satisfy formula ϕ minimally. Once we complete the update on all paths that violate $F\phi$, the resulting model will satisfy AFϕ, and is minimal from the original model (M, s).

Now let us consider the complexity of this update procedure. Clearly, for each such path, we need to update exactly one state, which costs time $\mathcal{O}(2^{|var(\phi)|})$. Given the tree-like counterexample (M, s) of formula $AF\phi$, generating a path of (M, s) and the update on the last state will cost $\mathcal{O}(|R| + 2^{|var(\phi)|})$ at most. Since there are at most $|S|$ states, we need to consider for the update the total cost will be no more than $\mathcal{O}(|R| \cdot |S| \cdot 2^{|var(\phi)|})$.

4. If (M, s) is a counterexample of A[ϕ U ψ] then there exists some path $\pi = [s_0, s_1, \ldots](s_0 = s)$ such that for some $s \in \pi$, $\pi \models \phi$ and $\pi \not\models \psi$ or $s \not\models \phi \vee \psi$. Here we have two possible cases for update.

Case 1. $(M, s)\pi \models \phi$ and $\pi \not\models \psi$. In this scenario the reader is referred to the update case for AFψ, as AFψ is equivalent to A[$\top$ U ψ].

Case 2. $(M, s)\pi \not\models \phi \vee \psi$. In this case we need to update the state where neither ϕ or ψ are satisfied. There exists two options, either satisfy ψ at the states and terminate, or satisfy ϕ at s, then at some future state update by A[ϕ U ψ].

For any state s where $(s, s) \in R$ a path π exists where $[s, s, \ldots]$ and s is traversed infinitely. This path needs to satisfy the subformula ψ and thus satisfy any path which exists from this state. Another condition on A[ϕ U ψ] is for leaf states. Any state s_j where $s_j \in \pi$, π is a finite path, $\pi = [s_0, \ldots, s_j]$ and there is no s_k such that $(s_j, s_k) \in R$ must be updated to satisfy ψ.

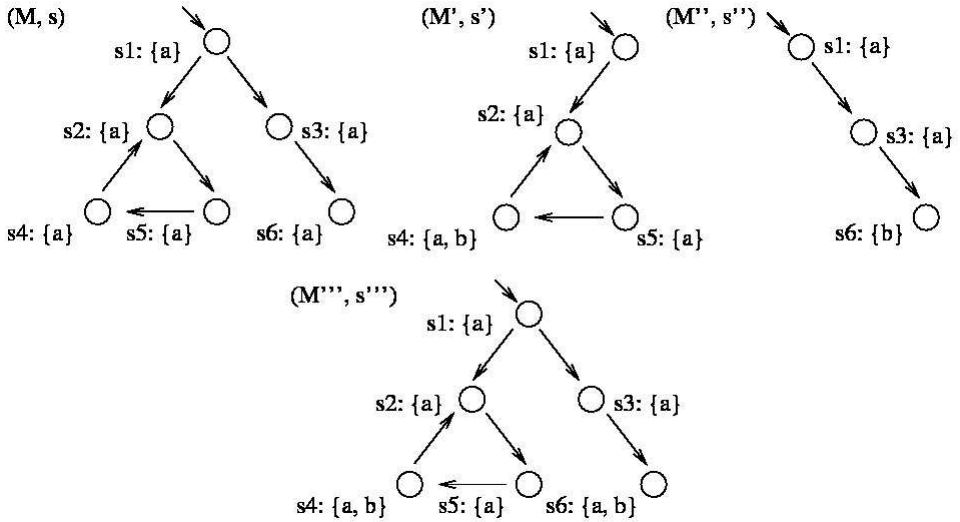


Figure 2.11: Possibilities for update for (M, s) by property A[a U b].

For each path we need to update at least once for each violation state that exists, costing $\mathcal{O}(2^{|var(\phi)|})$. Like AF, AU traverses a path from (M,s), updating states will cost $\mathcal{O}(|R| \cdot 2^{|var(\phi)|})$. As we need to consider $|S|$ states for update we get $\mathcal{O}(|S| \cdot |R| \cdot 2^{|var(\phi)|})$ for the resulting complexity.

5. If (M, s) is a counterexample of $\mathrm{AG}(\phi \rightarrow \mathrm{AF}(\psi))$, then there exists some path $\pi = [s_0, s_1, \ldots, s_j](s_0 = s)$ such that for some $s \in \pi, s \models \phi$ and from s there is no s_j where $s_j \in \pi, s_j \not\models \psi$.

In this case we apply the semantics given earlier for AG and AF. Two options are available to satisfy the formula. Satisfying $\neg\phi$ at s, or satisfying $\mathrm{AF}(\psi)$ at a state deeper in the model. As shown in *Case 1.* of *3.* the optimal choice for modification is at states deeper in the model.

As with AF, for each such path we need to update exactly one state, which costs time $\mathcal{O}(2^{|var(\phi)|})$. Given the tree-like counterexample (M, s) of formula $\mathrm{AF}\phi$, generating a path of (M, s) and the update on the last state will cost $\mathcal{O}(|R|+2^{|var(\phi)|})$ at most. Since there are at most $|S|$ states we need to consider for the update the total cost will be no more than $\mathcal{O}(|R| \cdot |S| \cdot 2^{|var(\phi)|})$. □

2.5 Theory of Constraint Automata

In the previous sections we have shown that the weak bisimulation based minimal change principle for tree-like model update is defined purely based on Kripke structures, no system constraints or other domain dependent information are considered in generating an update result. However, when we perform a model update, we may require this update not violate other specified system functions (*e.g.* breaking a deadlock should not violate a liveness property in a concurrent program). Further, even if an updated model satisfies our minimal change criterion (*i.e.* Definition 2.7), it may not represent a valid result under the specific domain context. For instance, as showed in Example 2.4, (M_1, s_1) is a minimal updated model to satisfy formula $\mathrm{AG}\neg x \vee \mathrm{AF}\neg y$. In practice, however, M_1 might not be acceptable if changing the variable x's value is not allowed in all states in model (M, s).

This motivates us to take relevant system constraints into account when we perform a model update. Besides logic based *system domain constraints*, which can usually be specified using ACTL (or CTL) formulas, there are some more complex constraints that are usually not expressible or difficult to be represented in the form

of ACTL (or CTL) formulas. For instance, constraints related to system actions cannot be directly represented by ACTL (CTL) formulas. In the following, we study two such typical constraints we use to guide update: *variable constraints* and *action constraints* related to expressing the underlying system behaviours. Following this we demonstrate the technique on a case of the dining philosophers problem using action constraints.

Given a set of propositional variables V and a set of system actions A, we define a *Variable Constraint Automaton* constructed from V and A to be a finite deterministic automaton $\mathcal{VC}(V, A) = (S, \Sigma, \delta, q_0, F, v)$, where $S \subseteq 2^V \cup \{v\}$ is the set of states, $\Sigma = A$ is the input action symbols, $\delta : S \times \Sigma \to S$ is the total state transition function, $q_0 \in S$ is the *initial state*, $F \subseteq S$ is the set of final states, and $v \in S$ is the unique violation state.

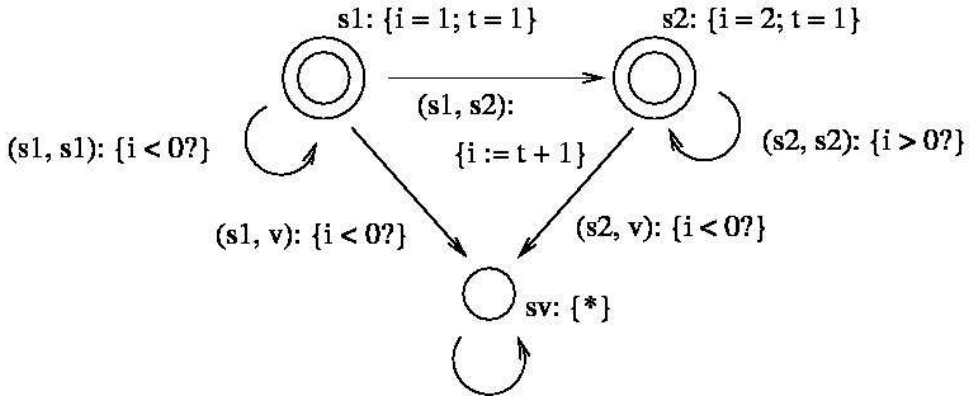


Figure 2.12: A variable constraint automaton.

A variable constraint automaton represents certain relations bound between a set of variables and a set of system actions. Consider two states $s_i,\ s_j \in S$, where s_i and s_j are not the violation state v, then a transition from s_i to s_j via action a: $\delta(s_i, a) = s_j$, indicates that by executing action a, variables' values represented by state s_i have to be changed to the corresponding variables' values represented by state s_j. Consider the variable constraint automaton depicted in Figure 2.12. Action $i := t + 1$ ties two variables i and t, so that i's value must depend on t's value when this action is executed[4]. On the other hand, action $i > 0?$ will not affect i and t's values, but execution of action $i < 0?$ will lead to the violation state v ($*$ represents any action symbols from Σ).

[4]For simplicity, we deliberately ignore the difference between a program and logic variable.

Similarly given a set of system actions A, we define an *Action Constraint Automaton* constructed from A to be a finite deterministic automaton $\mathcal{AC}(A) = (S, \Sigma, \delta, q_0, F, v)$, where $S \subseteq 2^A \cup \{v\}$ is the set of states, $\Sigma = \{preceded, next, exclusive\}$ is the set of input action constraint symbols, $\delta : S \times \Sigma \to S$ is the total state transition function, q_0 is the initial state, F is the set of final states, and v is the unique violation state.

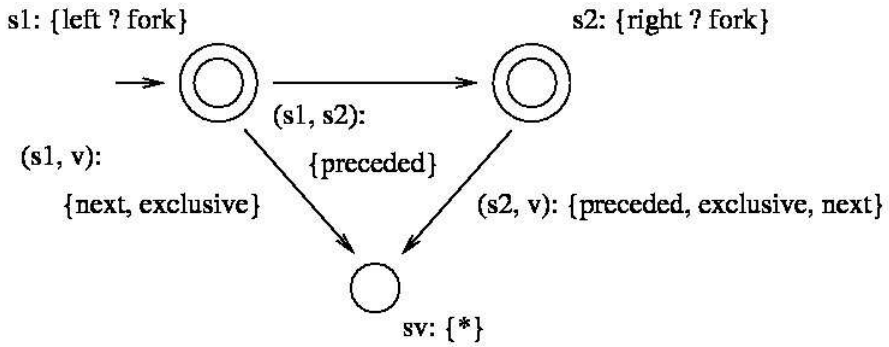
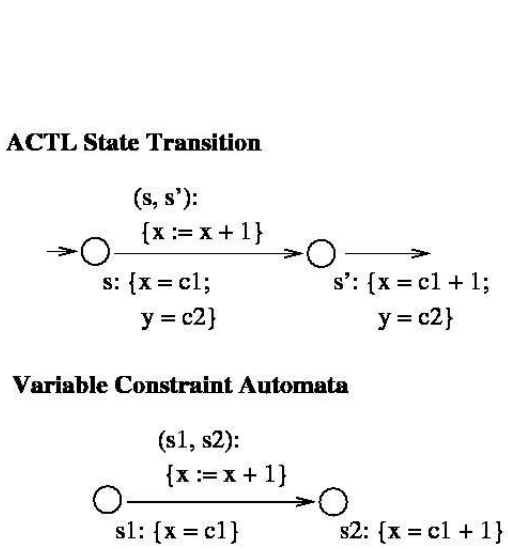


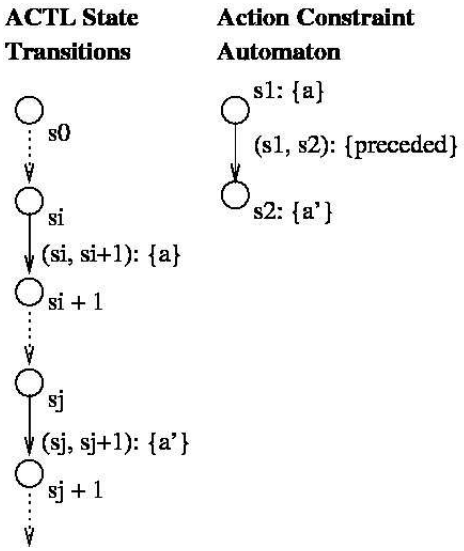
Figure 2.13: An action constraint automaton.

In an action constraint automaton, each state except the violation state is identified by a set of system actions. Then a transition between two states represents certain execution constraint between two specific sets of actions. For instance, if a_i and a_j are two system actions and states s_i and s_j are identified by actions $\{a_i\}$ and $\{a_j\}$ respectively, then $\delta(s_i, preceded) = s_j$ means that action a_i should be executed *earlier* than action a_j, $\delta(s_i, next) = s_j$ indicates that an execution of action a_i must *enforce* an immediate execution of action a_j, and $\delta(s_i, exclusive) = s_j$ states that an execution of action a_i must exclude a following execution of action a_j.

We give an example in Figure 2.13. In this case we have the two accepting states s_1 and s_2, the violation state v, and actions $left?fork$ and $right?fork$ at s_1 and s_2, respectively. An action constraint is created between s_1 and s_2 restricting the executions such that the action of state s_1 must come before the action of s_2, any other execution order causing the automata to transition to the violation state v.



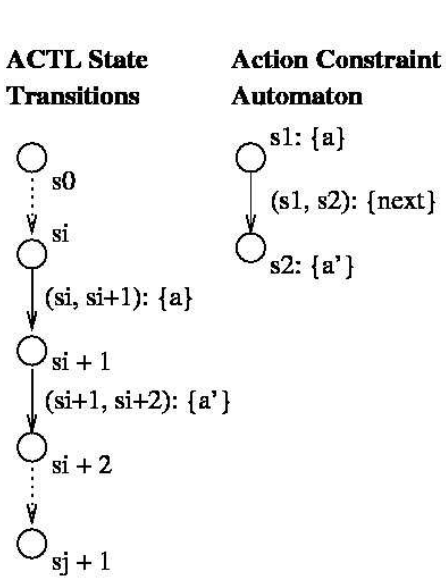
(a) State transition is consistent with variable constraints.



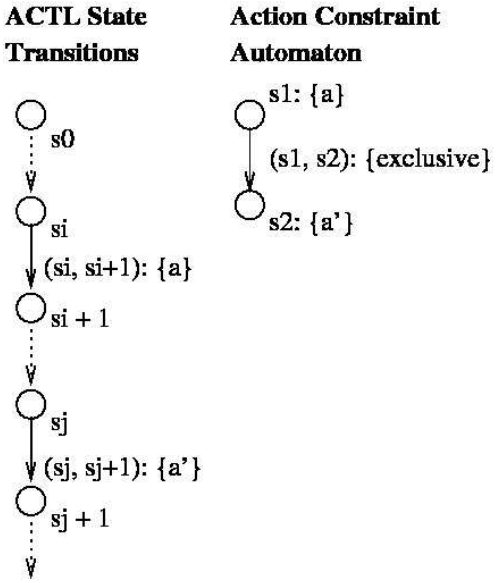
(b) State transition sequence complies to action constraint preceded.

Generally from a given system, we may generate more than one variable and action constraint automata. Now we may take the constraint automata into account when we perform model update. As discussed in the beginning of this section, in order to produce a more meaningful update result, our minimal change principle should be enhanced by integrating domain constraints, and system behaviour related variable and action constraints. Towards this aim, we first associate a set of system actions to a given Kripke structure. Recall that in the extent of model checking and model update, a Kripke structure actually represents the underlying system's behaviours where each state transition in the Kripke structure is caused by an execution of some system action. Therefore, for a given system, we can associate a set of system action A to the corresponding Kripke structure $M = (S,\ R,\ L)$ such that each state transition $(s,\ s') \in R$ is labelled by some action $a \in A$.

Definition 2.9. *Let $M = (S,\ R,\ L)$ be a Kripke structure, A a set of system actions associated to M, V a set of propositional variables, $\mathcal{VC}(V, A) = (S^{\mathcal{VC}}, \Sigma^{\mathcal{VC}}, \delta^{\mathcal{VC}}, q_0^{\mathcal{VC}}, F^{\mathcal{VC}}, v^{\mathcal{VC}})$ a variable constraint automaton, and $\mathcal{AC}(A) = (S^{\mathcal{AC}}, \Sigma^{\mathcal{AC}}, \delta^{\mathcal{AC}}, q_0^{\mathcal{AC}}, F^{\mathcal{AC}}, v^{\mathcal{AC}})$ an action constraint automaton. We say that M* complies to *$\mathcal{VC}$ and $\mathcal{AC}$, if the following conditions hold:*



(a) State transition sequence in (M, s) complies to action constraint *next*.



(b) Similarly, state transition sequence in (M, s) complies to action constraint *exclusive*.

1. *For each state transition* $\delta^{\mathcal{VC}}(s_1, a) = s_2$ *in* $\mathcal{VC}(V, A)$ *where* s_1, s_2 *are not the violation state, if there is a state* $s \in S$ *such that* $s_1 \subseteq L(s)$ *and* $(s, s') \in R$ *is labelled with* a*, then* $s_2 \subseteq L(s')$ *(i.e. variable bindings through action* a*);*

2. *For each* $\delta^{\mathcal{AC}}(s_1, \mathsf{preceded}) = s_2$ *in* $\mathcal{AC}(A)$*, where* s_1, s_2 *are not the violation state, for each* $a \in s_1$ *and* $a' \in s_2$*, there exists a path* $\pi = [s_0, \cdots, s_i, s_{i+1}, \cdots, s_j, s_{j+1}, \cdots]$ *in* M *such that* (s_i, s_{i+1}) *is labelled with* a *and* (s_j, s_{j+1}) *is labelled with* a' *(i.e.* a *occurs earlier than* a'*);*

3. *For each* $\delta^{\mathcal{AC}}(s_1, \mathsf{next}) = s_2$ *in* $\mathcal{AC}(A)$*, for each* $a \in s_1$ *and* $a' \in s_2$*, there exists a path* $\pi = [s_0, \cdots, s_i, s_{i+1}, s_{i+2}, \cdots]$ *in* M *such that* (s_i, s_{i+1}) *is labelled with* a *and* (s_{i+1}, s_{i+2}) *is labelled with* a' *(i.e.* a' *occurs next to* a*);*

4. *For each* $\delta^{\mathcal{AC}}(s_1, \mathsf{exclusive}) = s_2$ *in* $\mathcal{AC}(A)$*, for each* $a \in s_1$ *and* $a' \in s_2$*, there does not exist a path* $\pi = [s_0, \cdots, s_i, s_{i+1}, \cdots, s_j, s_{j+1}, \cdots]$ *in* M *such that* (s_i, s_{i+1}) *is labelled with* a *and* (s_j, s_{j+1}) *is labelled with* a' *(i.e.* a*'s execution excludes* a'*'s execution).*

Let us take a closer look at Definition 2.9. Condition 1 defines the transition function in a variable constraint automaton in the following way: if there is a state transition $(s, s') \in R$ in the Kripke structure, where some variables in s and s' are bound through action a, then in the variable constraint automaton, such variable bindings are represented through $\delta^{\mathcal{VC}}(s_1, a) = s_2$, where states s_1 and s_2 contain those corresponding variables respectively. Conditions 2, 3 and 4, on the other hand, define the transition function $\delta^{\mathcal{VC}}$ with respect to three specific constraints between actions in an action constraint automaton. In an action constraint automaton, a state is a set of system actions, and $\delta^{\mathcal{AC}}(s_1, \mathsf{preceded}) = s_2$ represents such a constraint that all actions in s_1 must be performed *before* actions occurring in s_2. Similar explanations are obtained for conditions 3 and 4 in the definition.

Given a set of domain constraints $\mathcal{C}$ (ACTL formulas) and a class of constraint automata $\Im$, we say that a tree-like model (M, s) *complies to* $\mathcal{C}$ and $\Im$ if $(M, s) \models \mathcal{C}$ and (M, s) complies to each constraint automaton in $\Im$. Now we can extend our previous tree-like model update with complying to domain constraints and constraint automata.

Definition 2.10 (Update complying to constraints)**.** *Let (M, s) be a tree-like model, $\mathcal{C}$ a set of ACTL formulas specifying the domain constraints, $\Im$ a class of system constraint automata, and ϕ a satisfiable ACTL formula such that $(M, s) \not\models \phi$. A tree-like model (M_1, s_1) is called a* result of updating (M, s) with ϕ complying to $\{\mathcal{C}, \Im\}$*, iff*

1. *$(M_1, s_1) \models \phi$ and complies to $\mathcal{C}$ and $\Im$;*
2. *there is a weak bisimulation H_1 between (M, s) and (M_1, s_1) such that there does not exist another tree-like model (M_2, s_2) satisfying that $(M_2, s_2) \models \phi$, (M_2, s_2) complies to $\mathcal{C}$ and $\Im$, and a weak bisimulation H_2 between (M, s) and (M_2, s_2) such that $H_2 < H_1$.*

We should indicate that combining domain constraints and constraint automata into our tree-like model update approach does not significantly increase the approaches complexity, remaining co-NP-complete.

2.6 Summary

In this chapter we have thoroughly investigated the preliminary concepts for effecting efficient satisfaction of some formal ACTL property in a local region of a Kripke structure. This included the conditions in which we can confidently say one new model is closer to the original model than some alternative, while still satisfying a property using weak bisimulation semantics as a way to determine ordering between different modification approaches. Further, we expressed the link between our approach and a previous approach for belief update by Katsuno and Mendelson, and showed how we can determine if applying some modification using one property will interfere with the satisfaction of some secondary desired property. We also analysed computational properties, looking at expected time complexities for generating candidate local model updates using common properties. Finally, we extended the theoretical approach with constraint automata as a means of improving the quality of the update process without significantly increasing complexity.

In the following chapter we bridge the gap between theory and implementation by generating algorithms enacting property satisfaction on concrete system model counterexamples, guided by the characterisations in this chapter. These are the preliminary steps towards meeting the goal of developing software which automatically performs localised update.

Chapter 3

Algorithms for ACTL Local Model Update

3.1 Basic Idea for Algorithm Design

In the prior chapter we investigated theoretical foundations of local model updates. The theory has been used as a framework to generate algorithms that can be used as modules for the system implementation. The proposed algorithms in this chapter enact the principles outlined in the formal framework, including weak bisimulation based minimal change and the primitive update archetypes. The update algorithm works as a search procedure, taking the local model to be updated, the initial state and an ACTL formula, and processes the model based on the ACTL formula token semantics. Finally, the algorithm returns a set of possible modifications which satisfies the model when applied. The algorithm is also built with design patterns used from previous implementations of SAT solvers and model checkers [66, 101].

Although we have analysed the semantic characterisations of local model update in the prior chapter, many challenges exist in translating characterisations to algorithms. One challenge in deriving satisfying modifications to a model is that to satisfy a temporal property, updates need to be enacted over model path regions which are not the state it is satisfying. An example of this is satisfying AF$a \wedge$ AF$\neg a$ at s. It is necessary to satisfy a and $\neg a$ at all connected branching paths from s, but not at the same states. This is covered with persistence in Section 2.3, it seems logical to satisfy these two propositions sequentially but it is difficult to tell if this technique would scale with more propositional atoms.

Substituting a state with another which satisfies another propositional atom can lead to inconsistencies in the satisfaction of some property. Another possibility exists with formulae where two states require satisfaction by some subformula and can be satisfied by one minimal modification (*e.g.* $\mathrm{AG}((c \rightarrow \mathrm{AF}a) \wedge (d \rightarrow \mathrm{AF}a))$). To handle these cases, we look to the characterisations and minimal change based on bisimulation semantics for heuristics, and to derive coherent updates. It can also be seen that performing comparisons between entire local models can be computationally expensive. A shorthand system for representing changes to a local model is given which represents differences between models in an atomic manner using update tuples. Update tuples are maintained in sets of updates, such that multiple modifications can be performed and comparisons can be made to determine minimality of change. This is discussed further in the chapter.

Other points exist which should be addressed about the implementation; for efficiency and completeness purposes we transform the repair update problem to a search procedure over the size of the local model region using a combination of iterative and recursive path traversal methods in conjunction with the semantics of ACTL. By doing this we can efficiently search over the model using the derived characterisations and build a set of possible atomic updates that we can apply to the local region of the model to effect satisfaction. Also note that the update search procedure does not actually modify the underlying Kripke model; control passes along the model state paths based on the formula semantics and functions return enumerated sets of atomic update tuples which satisfy the model by the formula. Included with the update search algorithm is the function which applies the update once determined. When the minimal possible updates have been determined it may be returned. Next, implementing the most minimal update requires accessing the referenced state(s) or relation(s) and applying the atomic update. All algorithms in the following section are designed in pseudo-code for readability and to give focus to important design elements.

To demonstrate, we give an example illustrating the derivation of an update over a model not satisfying a compound temporal property $\mathrm{AXAF}a$.

Example 3.1. *Consider the model represented in the transition graph in Figure 3.1 where* $M = (S,\ R,\ L)$, $S = \{s_1,\ s_2,\ s_3,\ s_4,\ s_5,\ s_6,\ s_7\}$, $R = \{(s_1,\ s_5),\ (s_5,\ s_6),\ (s_1,\ s_2),\ (s_2,\ s_5),\ (s_5,\ s_3),\ (s_2,\ s_3),\ (s_3,\ s_4),\ (s_4,\ s_7),\ (s_7,\ s_3)\}$ *and* $L(s_1) = \{a\}$, $L(s_5) = \{a,\ b\}$, $L(s_2) = L(s_3) = L(s_4) = L(s_6) = L(s_7) = \{b\}$. *Suppose it is required that* $(M,\ s_1)$ *satisfy the property* $AXAFa$. *Passing this into a model checker we find* $(M,\ s_1) \not\models AXAFa$ *witnessed by the local model* $C = (S,\ R,\ L)$, *where* $S = \{s_1,\ s_2,\ s_3,\ s_4,\ s_7\}$, $R = \{(s_1,\ s_2),\ (s_2,\ s_3),\ (s_3,\ s_4),\ (s_4,\ s_7),\ (s_7,\ s_3)\}$ *and* $L(s_1) = \{a\}$, $L(s_2) = L(s_3) = L(s_4) = L(s_7) = \{b\}$, $L(s_5) = \{a,\ b\}$.

With this local model we start from the initial state s_1 *and apply the semantics of* $AX\phi$ *to route update.* AX *semantics requires that all states immediately related to the current state satisfy the sub-formula, meaning update needs to occur from* s_2 *using* AFa. *From* AF *semantics we know that from the current state all paths have to at some point satisfy the subformula, in this case* a. *Following the local model path, we find that the deepest depth ends in a cycle, we can modify some state* $s_3,\ s_4$ *or* s_7 *to satisfy* a *and thus the formula as a whole. Being a cycle, the characterisations imply that the most appropriate update would be to update the model by substituting state* s_3 *with another state* s_3', *where* s_3' *satisfies* a. *This is represented using an update token set* $\mathcal{U} = \{(s_3,\ +,\ a)\}$, *indicating update at state* s_3 *by adding the label* a *to the state. We go further into update tuples in Subsection 3.2.2 of this chapter.*

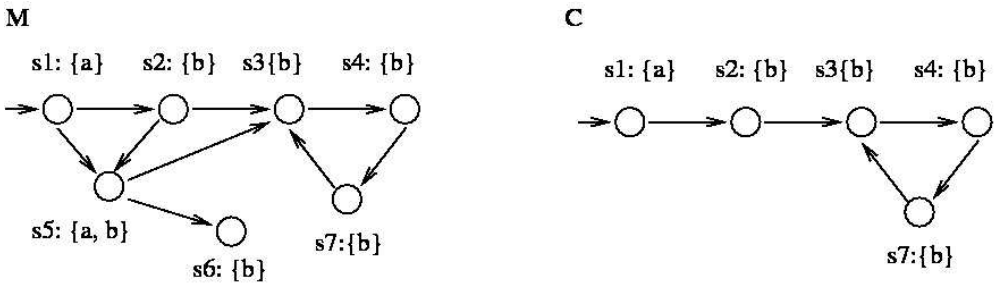


Figure 3.1: Example where $(M,\ s) \not\models$ AXAFa.

From this example, we can see that enacting update proposes several challenges. This includes separation of duty for each ACTL token based on their specific semantics, and methods for traversing the Kripke model paths. Beyond this example, we require means of determining which update is more minimal and how to determine persistence between two updates, which we will analyse later in this chapter. In the following section we will analyse and construct the algorithm used to enact update.

3.2 Algorithms

3.2.1 Main Update Algorithm

$Update_c$ in Algorithm 3.1 is the main update algorithm, where c is used to indicate it as the core function[1]. To begin an update session, we pass as argument a local model, an ACTL formula, and an initial state to $Update_c$. $Update_c$ is used to route the applied semantics to the model based on the root formula token in the ACTL formula[2].

In each of the temporal operations the model is traversed based on the requirement to determine satisfaction of the formula at different states. As this is a recursive procedure, once the function is called to handle the update case returns, the update set(s) will be aggregated and control will be returned to the calling function. This function is called by the other sub-functions as a means of recursively determining satisfaction of the sub-formulas it is passed. In this way calls pass through the sub-formulas of the property depth-first through the tree-like structure, where the leaves of the formula tree are the propositional atoms. It follows that the nesting depth of the formula will determine the recursion stack call size in implementation. When propositional atoms are encountered in the formula, $Update_p$ is called and methods for substituting a state with another state which satisfies the required propositional atom are ascertained and returned in the form of an update tuple u.

[1] In this chapter update functions are subscripted with the formula token to maintain consistency.

[2] See Definition 2.1 for ACTL syntax.

Algorithm 3.1: $Update_c(M,\ s,\ \phi)$.

Input: $M = (S,\ R,\ L)$, where $s \in S$ and $(M,\ s) \not\models \phi$;

Output: Update set $\mathcal{U}$, where $u \in \mathcal{U}$ and $u = (s|r,\ +|-,\ p)$, $s|r$ is the state or relation modified, p the related atomic proposition, and $+|-$ the update modifier.

```
01: Update_c(M, s, φ)
02: {
03:     case
04:         φ ≡ p or φ ≡ ¬p:    return Update_p(M, s, φ);
05:         φ ≡ φ1 ∨ φ2:        return Update_∨(M, s, φ);
06:         φ ≡ φ1 ∧ φ2:        return Update_∧(M, s, φ);
07:         φ ≡ AG φ1:          return Update_AG(M, s, φ);
08:         φ ≡ AF φ1:          return Update_AF(M, s, φ);
09:         φ ≡ AX φ1:          return Update_AX(M, s, φ);
10:         φ ≡ AU φ1:          return Update_AU(M, s, φ);
11: }
```

This recursive design pattern chosen is modular, efficient, facilitates integration into future update tools, is appropriate for ACTL property structure and also legible to the point of facilitating understanding for future researchers. To give a deeper insight into the link between formula parsing and applying update semantics to the Kripke model based on which formula tokens are being parsed, we use Example 3.2 to explain program function flow routing. For more information about this form of program flow see [40, 66] for similar designs for both checking and update.

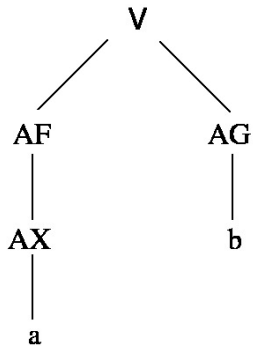


Figure 3.2: The parse tree for ACTL formula (AFAXa)$\vee$ AGb.

Example 3.2. *Consider the ACTL formula $\phi \equiv (AFAXa) \vee AGb$ represented by the parse tree in Figure 3.1. Here the root of the formula is $\vee$ and its two arguments are $AFAXa$ and AGb. In the algorithm $Update_c$ identifies the root of the formula and passes control to the disjunctive semantics function $Update_\vee$[3]. $Update_\vee$ requires the return values of its two arguments to apply semantics and return the best update, for each argument we call $Update_c$. $Update_c$ is called firstly with M, s and $AFAXa$. $Update_c$ then identifies AF as the current formula and routes execution to $Update_{AF}$. $Update_{AF}$ then applies its semantics to determine what will satisfy AXa at some future state in the model. This requires calls to $Update_c$ to determine how to satisfy the subformula in conjunction with the semantics of AF. $Update_c$ then routes execution to $Update_{AX}$ with the current state s and model M. $Update_{AX}$ applies its semantics on the given state and to determine satisfaction calls $Update_c$ to route control for the propositional case, $Update_p$ for a.*

Once update has been ascertained for the model region from $Update_p$, the update tuples representing the minimal update are returned as an update set $\mathcal{U}$ back through the calling functions finally to $Update_\vee$. This repeats for its other argument by passing the formula AGb to $Update_c$ with M and the current state s. With both possible update sets returned, $Update_\vee$ returns the most minimal update based on heuristics guided by weak bisimulation principles, explained further in this chapter.

3.2.2 Update Tuples

In the previous iteration of model update, the algorithm worked directly on the model and applied modifications in real time, returning the resulting model satisfying the ACTL property. In this process, however, applying the update is linear in time relative to the count of atomic updates. Comparing this to the search procedure for a set of atomic updates which satisfy the property, the cost is minimal. The algorithm returns sets of update tuples which indicate what aspects of the model to modify, to enact property satisfaction in the model by applying the necessary update. This allows the decoupling of the update search procedure from application

[3]Specific semantics of each update handler function are explored in following sections of this chapter.

of modification and to allow rollback of modifications given the update tuples and updated model.

Definition 3.1. *An update tuple u is a 3-ary tuple $(s|r,\ -|+,\ p)$ where*

1. *$s|r$ is a state $s \in S$ or relation $r \in R$;*
2. *$-|+$ is the removal/addition modifier, indicating whether the state, relation or label is being added or removed;*
3. *p is the propositional atom being added or removed at a given state, this value is left empty ($Null$) if the modification is adding or removing a state or relation.*

Update tuples are a useful notation which allows us to express a modification to a Kripke structure (*e.g.* $(s, +, Null)$ indicates an added state s). We can use the notation $u[0]$ to represent the first argument and $u[1], u[2]$ to represent the second and third, respectively.

It is also important to note that the first argument of an update tuple will always be some state or relation that exists within the local model in question. This means an update tuple can be determined to be pertaining to a state or a relation by performing a check to see if it is an element of S or R, respectively.

In general cases more than one atomic modification needs to occur to satisfy a property formula. For this reason functions return sets of updates $\mathcal{U}$, such that each update is applied to make the necessary change to satisfy the property. Once the main function returns an update set each tuple can be used to perform the actual modification. We can ascertain the cardinality of changes to a model through the short hand $|\mathcal{U}|$.

This method of representation of update relative to some model has many advantages. Primarily it can be seen as a method for determining difference between two models relative to the original by looking at the update set instead of having to decompose the updated model such that the difference between the original and some other different update could be seen[4]. Further to this, it allows a more efficient

[4]This was listed as one of the limitations to the *Model Update* approach used by Ding in her Thesis *Model Update for System Modifications*.

representation footprint for large updates, as the difference between models is kept instead of the entire updated model. Another advantage of this approach derives from the fact that different updates may be required to be performed at the same state by the same modification or contain updates which conflict. Using this representation of updates, these checks can be done in constant time with the appropriate data structures. The approach also allows heuristics to be generated which can be used to approximate bisimulation ordering, giving a computationally faster method of determining which is a more minimal update.

3.2.3 Propositional Atom Update

The idea behind atomic propositional update shown in Algorithm 3.2 is based on previous state substitution methods such as operator PU3 in [101] and adheres to the characterisations given in the previous chapters. The algorithm for the update of a single state's label function by a propositional atom is the base case presented for possible update types. In this approach, updating by higher level nested temporal tokens will occur first on the model, going through several layers of nested formula operations and applying the required model routing. This is done before applying state substitutions by using the function $Update_p$. In this way updating a state label is functionally equivalent to substituting the state with another that has the same labels, but satisfies the propositional atom.

This function handles both negated and positive atom formula types, based on the notion that ACTL formulas require that negation only be applied to propositional atoms. A separate update function for handling negation is not required as one of the initial preconditions for each ACTL formula is that negation be restricted solely to propositional atoms. Before update however, the ACTL formula is checked to satisfy this condition, and if found to have negation outside of propositional atoms equivalences are used to attempt to transform it to ACTL. If no ACTL equivalence can be found an exception is raised.

The main intuition of the function is to check the sign of the atom (*lines 3, 5, 7 and 9*) and the presence of the corresponding label at the given state's label set, then based on this, return the update tuple for substituting the state. The new state would satisfy the given formula (*i.e.* $u = (s,\ +,\ p)$ for adding a label p at state s, and removal $u = (s,\ -,\ p)$).

For a positive propositional atom, if the corresponding label is present in the label set associated with the state, then the state satisfies the property and a null update is returned. This signifies that the state satisfies the atom and no update is required (*line 11*). In this sense, we can say that the update process is *idempotent* (*i.e.* if an attempt is made to update a model by a specification it already satisfies or was previously updated to satisfy by there is no effect to the underlying model). Otherwise, if the formula is a positive propositional atom and the corresponding label is not present in the state's labels, we simulate an update by unioning the label set of the state with the new atom. We represent this formally on the state s by noting that $s \not\models p$ and creating a new state s', where $L(s') = L(s) \cup \{p\}$ and $s' \models p$. The function returns an update tuple $u = (s,\ +,\ p)$ to indicate that this unioning is necessary for atom satisfaction (*line 9*). This works for a state's label space, in that absence of a label in the label set represents the negation of the atomic proposition in ACTL property semantics.

Example 3.3. *In Example 3.1 we defined s_3 with the label function $L(s_3) = \{b\}$, if we apply $Update_c(M, s_3, a)$ to have s_3 satisfy the property a. $Update_c$ routes execution to the function $Update_p$ which checks the label function of the state, and based on the negation status of the formula atom and the presence or absence of the atom in the label, returns the corresponding update tuple indicating what to add or remove from the label function. The algorithm will note the absence of the label and the lack of negation, indicating the property can be satisfied by modifying the label function to satisfy a such that $L(s_3) = \{a,\ b\}$. $Update_p$ indicates this by returning the update tuple set $\mathcal{U} = \{u = (s_3,\ +,\ a)\}$. Similarly, if the label function of s_3 was originally $L(s_3) = \{a,\ b\}$, $Update_p$ would return the empty update set ($\emptyset$).*

The secondary case which exists for $Update_p$ is where the property is a negative propositional atom. With the given state we check to see if the atom is an element of the label set for the state. If not, we return the empty update to indicate the absence of the atom at the state and thus that $M, s \models \neg p$ (*line 11*). Otherwise the atom is present in the label set of s and the set difference is taken to remove p to satisfy the property (*line 8*). Formally we say $s \not\models \neg p$ and create s' such that $L(s') = L(s) - \{p\}$ and $s' \models \neg p$. The function $Update_p$ returns the update tuple set $\mathcal{U} = \{(s,\ -,\ p)\}$ to indicate that state substitution needs to be performed.

The updated model M' comes about through the modification of states, labels and transitions on M inherent with performing an update. This has the consequence of having $M' \models \phi$. In the formalism, when replacing a state s with another s' which satisfies the property we do a set union of the state set with s' and set difference against s to derive the new state set S'. This is represented formally as $S' = (S \cup \{s'\}) - \{s\}$. To obtain the updated set of relations we perform set union on all relations in the relations where s' is in the domain or range and it's associated state s is in the domain or range and remove all relations in the domain and range of the state s, *i.e.* $R' = R - \{(x, s), (s, y) | (x, s) \in R, (s, y) \in R\} \cup \{(x, s'), (s', y) | (x, s) \in R, (s, y) \in R\}$. L' is derived by applying the set union to the set of label functions to include s' and the set difference to remove the function which maps s to its label set. However, in practice each method returns an update tuple set representing the update that needs to occur to satisfy the sub-property at the state.

Another two cases for $Update_p()$ involves processing tautology ($\top$) and contradiction ($\bot$) formulas by returning the update it would represent. If a tautology is received, an empty update tuple is returned as the model vacuously satisfies the symbol(*line 4*). If a contradiction is given, a special value MAX is returned, meaning the worst case tuple size (*line 6*). If any comparison is made in the algorithm between two update tuple sets, the update tuple set without the contradiction signal MAX will always be seen as the preferred choice and returned. Similarly, if a comparison is made with an update tuple satisfying a tautology, the tautology will be selected.

As mentioned earlier $Update_p$ is the most concrete level of update and is called whenever it is necessary to substitute a state with a new state which satisfies the propositional atom in question. As ACTL allows nested temporal operations, functions used for update based on temporal properties call $Update_c$ when a state needs to be updated by some sub-formula, be it nested formula or atomic. $Update_c$ routes it to the appropriate function specific to the ACTL token, propositional atoms modifications get routed to $Update_p$.

Algorithm 3.2: $Update_p(M,\ s,\ \phi)$.

Input: $M = (S,\ R,\ L)$, where $s \in S$.

Output: Update tuple $u = (s|r,\ +|-,\ p)$, where $s|r$ is the state or relation modified, p the related atomic proposition, and $+|-$ the update modifier.

```
01: Update_p(M, s, φ)
02: {
03:     if φ ≡ ⊤:
04:         return ∅;
05:     else if φ ≡ ⊥:
06:         return {MAX};
07:     else if φ ≡ ¬p and p ∈ L(s):
08:         return {u = (s, −, p)};
09:     else if φ ≡ p and p ∉ L(s):
10:         return {u = (s, +, p)};
11:     return ∅;
12: }
```

3.2.4 Update Application

As mentioned earlier, having found a set of modifications which when applied satisfies the property in the most minimal sense, the application of the update is trivial. $Update_{apply}$ takes as argument the local model to be updated and the set of update tuples $\mathcal{U}$ which contains the possible types of modification on M and applies the updates based on the tuple arguments. $Update_{apply}$ applies addition and removal of states, addition and removal of relations and the substitution of states with new states which satisfy propositions. Once the update tuple set has been exhausted and all updates have been applied, $Update_{apply}$ returns the modified local model $M' = (S', R', L')$.

Example 3.4. *Consider the scenario where we have the model M from Example 3.1 and some set of update tuples $\mathcal{U} = \{((s_7, s_3), -, Null),\ (s_7, -, Null),\ (s^*, +, Null),\ (s^*, +, c),\ ((s_4, s^*), +, Null)\}$. We execute $Update_{apply}$ with the arguments M and $\mathcal{U}$. The first modification is $((s_7, s_3), -, Null)$, indicating the removal of the relation (s_7, s_3) from the relation set. The element type of $u[0]$ and addition or removal modifier $u[1]$ at lines 16 and 19 enables the modification of the relation set R with*

Algorithm 3.3: $Update_{Apply}(M,\ s,\ \mathcal{U})$.

Input: $M = (S,\ R,\ L)$, $s \in S, \mathcal{U}$, where $u \in \mathcal{U}$ and $u = (s|r,\ +|-,\ p)$, where $s|r$ is the state or relation modified, p the related atomic proposition, and $+|-$ the update modifier.

Output: Updated Model M', where $(M',\ s') \models \phi$.

```
01: Update_Apply(M, s, U)
02: {
03:     for u in U :
04:         if u[2] != Null //label satisfying substitution case
05:             if u[1] == + :
06:                 L(s') = L(u[0]) ∪ {u[2]}; //u[0] = s, u[2] = p
07:             if u[1] == - :
08:                 L(s') = L(u[0]) − {u[2]};
09:             S' = S ∪ {s'};
10:             R' = R − {(x,s),(s,y)|(x,s) ∈ R,(s,y) ∈ R} ∪
                    {(x,s'),(s',y)|(x,s) ∈ R,(s,y) ∈ R};
11:         if u[0] ∈ S //case of state modification
12:             if u[1] == +:
13:                 S' = S ∪ {u[0]}; //u[0] = s
14:             if u[1] == −:
15:                 S' = S − {u[0]};
16:         else u[0] ∈ R //case of relation modification
17:             if u[1] == +:
18:                 R' = R ∪ {u[0]}; //u[0] = (s_i, s_j)
19:             if u[1] == −:
20:                 R' = R − {u[0]};
21:     return (M', s');
22: }
```

$R' = R - \{u[0]\}$ at line 20. The next updates $(s_7, -)$ and $(s^*, +)$ are identified as state modification tuples at line 11 checking $u[0]$. Enacting these modifications to the state set are the operations $S' = S - \{u[0]\}$ and $S' = S \cup \{u[0]\}$ at lines 15 and 13, respectively. Next, the update $((s_4, s^*), +, Null)$ is handled similarly to the relation removal case. Line 16 identifies this as pertaining to relations and line 17 identifies the addition modifier. With this, the operation $R' = R \cup \{u[0]\}$ is performed, updating the relation set. Lastly, the update $(s^*, +, c)$ performs an operation functionally equivalent to modifying s^* to satisfy the label c. A new label

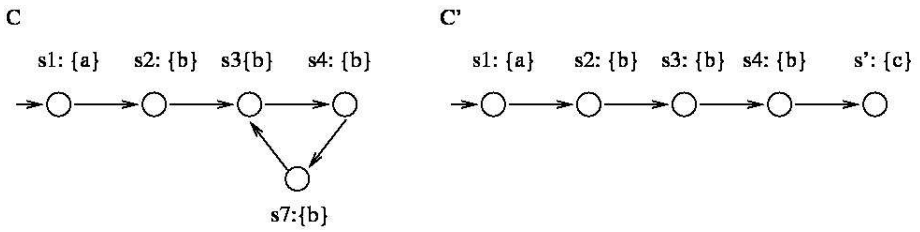


Figure 3.3: The resultant updated local model after $Update_{apply}$ is executed.

function is created for s^*, $L(s^*) = L(u[0]) \cup \{u[2]\}$ *at line 6. After this if* s^* *is a new state it is added to the state set, otherwise the operation is idempotent and introducing the state will not re-add the state. The relations of the replaced states are routed to the new state and removed from the replaced state. Finally, the resultant updated local model derived can be seen in Figure 3.3.*

3.2.5 Conjunctive Formula Update

Formulas based on conjunction token $\wedge$ are considered compound formulas and are similar to the base propositional case. In Algorithm 3.4, the list of sub-formula arguments of the conjunctive formulas are aggregated (*line 4*). Updates are performed by calling $Update_c$, using each sub-term and the set of updates required to satisfy both sub-properties of the conjunction are returned to the calling function.

To determine if the two subformulas have no conflicting modifications after the update process, we apply the modification to a copy of the original. We return these updates as a conjunctive set of necessary update tuples (*i.e.* each of these updates need to occur to satisfy the property).

What has occurred in this function is the processing of the model to discover which will satisfy each argument of the conjunction, but it is often the case that these recommended changes are not compatible, even though the formula as a whole is satisfiable. Consider the property AF$a\wedge$ AF$\neg a$ applied to $(M,\ s_6)$ in Example 3.1. To obtain the minimal update we substitute s_6 with a state that satisfies both a and $\neg a$, which is not possible. Conflicts of this form can be identified by unioning the

Algorithm 3.4: $Update_{\wedge}(M,\ s,\ \phi)$.

Input: $M = (S,\ R,\ L),\ \phi,\ s$, where $s \in S$ and $(M, s) \not\models \phi$.

Output: Update set $\mathcal{U}$, where $u \in \mathcal{U}$ and $u = (s|r,\ +|-,\ p)$ where $s|r$ is the state or relation modified, p the related atomic proposition, and $+|-$ the update modifier.

```
01: update∧(M, s, φ) :
02: {
03:     for φi in φ0 ... φn :
04:         U = U ∪ Update_c(M, s, φi);
05:     if Update_apply(M, s, U) ⊭ φ :
06:         return −1;
07:     return U;
08: }
```

returned update tuples and identifying updates which conflict with one another.

Conflicts can be defined as two or more opposing update tuples which require the addition and removal of a state or relation, or the substitution of a state by another which satisfies and doesn't satisfy some label. Further, update conflict arises when some state or relation is to be removed from the model, based on one argument but the other argument requires addition of some element in the path where the removed element existed. This relates to the problem of persistence between properties in Subsection 2.2 of the former chapter, in that persistence is the characteristic of the model maintaining the satisfaction of another earlier property, whereas in this case we wish to maintain the persistence of two properties being concurrently applied, which may require conflicting modifications. If a conflict is found between two update tuples and there exists other updates which do not cause conflict and satisfy the property, these can be returned. To guarantee the model satisfies both sub-formulas $Update_c$ is re-called once the valuation has been applied with $Update_{apply}$ such that we can ascertain no conflict exists.

3.2.6 Modelling a Minimal Change Heuristic

From the previous characterisations, we can see that for modifying model regions, weak bisimulation semantics tells us that an updated model and the original model will be more similar than the original and some other updated model if the difference between the models is at leaf states, or at greater depths of the tree-like model, for any given possible modification to the underlying model. In local model update there is no notion of preference between adding or removing a state, relation or label. Each type of update is assumed as equal in its ability to lead the model towards satisfying a property. We can compare these equivalent atomic updates against one another by comparing depth of modification relative to the initial root state.

Firstly, we note that if for two update sets $\mathcal{U}_1, \mathcal{U}_2$, if $\mathcal{U}_1 \subset \mathcal{U}_2$, it is reasonable to assume the subset of the two being a better update of the two. Either will satisfy the property, but the subset will necessarily be minimal. Next, in the case where multiple updates are required at different depths, we apply the update depth heuristic to determine which collection of updates is optimal.

This notion of update ties in with weak bisimulation semantics from Definition 2.5 in that it is a process of mapping branching similarities between some original model and an admissible model which satisfies the property through minor modifications. The differences between the original model and its admissible candidate exist as the update tuple set, describing the steps needed to satisfy the property. With two update tuples we can determine ordering by allowing preference to modifications which occur more regularly deeper.

To quantitatively compare the relative depths, we define $depth(\mathcal{U})$ as the set of cardinal depths of states in M, where an update is occurring with respect to the model, or $depth(\mathcal{U}) = \{depth(M, s) | \exists (s, +|-, p) \in \mathcal{U}\}$. We define $depth(M, s)$ as the individual depth of s with respect to M, such that it is the number of relations in the path connecting root s_0 to s in M. We can compare integer depth sets relative to the root using the operator $\prec$, where

$depth(\mathcal{U}_1) \prec depth(\mathcal{U}_2) \stackrel{def}{=} \forall d_1 \in depth(\mathcal{U}_1)\ \exists d_2 \in depth(\mathcal{U}_2)$, such that $d_1 < d_2$.

Using this we can match each depth d in one update depth set to the other. If there exists some greater depth d for each depth in the first set, we say $\mathcal{U}$ is more minimal from weak bisimulation semantics. Finally, if all metrics for minimal update are otherwise equal we can use the cardinality of update tuple sets $\mathcal{U}_1$,$\mathcal{U}_2$ to finally decide which is the more minimal (*i.e.* $|\mathcal{U}_1| < |\mathcal{U}_2|$).

Algorithm 3.5: $minChange(\mathcal{U}_1,\ \mathcal{U}_2)$.

Input: $\mathcal{U}_1,\ \mathcal{U}_2$, where $\mathcal{U}_1, \mathcal{U}_2$ are update sets, and $u \in \mathcal{U}$, where $u = (s|r, +|-, p)$ and $s|r$ is the state or relation modified, p the related atomic proposition, and $+|-$ the update modifier.

Output: $\mathcal{U}$, where $u \in \mathcal{U}$ and $Update_{apply}(M, s,\ \mathcal{U}) \models \phi$.

```
01:  minChange(U1, U2) :
02:  {
03:      if U1 ⊂ U2 :
04:          return U1;
05:      if U2 ⊂ U1 :
06:          return U2;
07:      else if depth(U2) ≺ depth(U1):
08:          return U1;
09:      else if depth(U1) ≺ depth(U2):
10:          return U2;
11:      else if |U1| < |U2|:
12:          return U1;
13:      else:
14:          return U2;
15:  }
```

Example 3.5. *Application of* $Update_c()$ *to the model* M*, represented as the transition graph in Figure 3.4 has determined three update sets which will satisfy the property* ϕ *if applied to* (M, s)*. These are* $\mathcal{U}_1 = \{(s_4, -, y),\ (s_3, -, y)\}$ $\mathcal{U}_2 = \{(s_4, -, y),\ (s_5, -, y),\ (s_6, -, y)\}$ $\mathcal{U}_3 = \{(s_2, -, y),\ (s_5, -, y),\ (s_6, -, y)\}$*. To ascertain the better update we determine the depths of the update states relative to the model's root state using* $depth()$*. We find* $depth(\mathcal{U}_2)$*, to be the most minimal change based on prior semantics.*

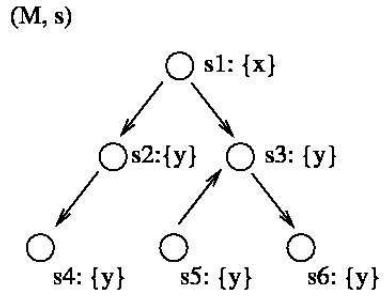


Figure 3.4: Transition graph of $(M,\ s)$ in Example 3.1.

3.2.7 Disjunctive Formula Update

The disjunctive update function $Update_{\vee}$, described in Algorithm 3.6 operates similarly to the conjunctive case. The set of sub-formula arguments are aggregated and update is applied using the $Update_c$ call mentioned earlier. In this case however only one update set needs to be returned to the calling function, the update which will create a model closest to the original based on weak bisimulation ordering. For this reason a function which determines the more minimally modifying update set between multiple options is necessary. This is implemented in disjunction and will be described in the subsequent section.

Algorithm 3.6: $Update_{\vee}(M,\ s,\ \phi)$.

Input: $M = (S,\ R,\ L),\ \phi,\ s$, where $s \in S$ and $(M, s) \not\models \phi$;

Output: Update set $\mathcal{U}$, tuple $u = (s|r,\ +|-,\ p)$, where $s|r$ is the state or relation modified, p the related atomic proposition, and $+|-$ the update modifier.

```
01: Update∨(M, s, φ) :
02: {
03:     U = Update_c(M, s, φ_1);
04:     for φ_i in φ_2 ... φ_n :
05:         U = minChange(Update_c(M, s, φ_i), U);
06:     return U;
07: }
```

For each disjunctive sub-formula given in the property, a call to $Update_c$ is made with the model and current state as argument. This returns an update set $\mathcal{U}$,

which can be compared against the other generated update sets based on the weak bisimulation heuristics and determine which is closest to the original model. This is done by generating the first update set as a basis of comparison as optimal, then iterating through the other sub-formulas and comparing the new formula against the current most optimal update set. If the minimal change heuristic finds the newly generated update set to be closer to the original than the current optimal update set, the new update set is set as the most optimal update. Once each sub-formula has been tested, the remaining update set will be the update which satisfies the property and is closest to the original model. This function accepts n formulas, such that an arbitrary amount of formula arguments in the disjunction can be accepted.

In the case that some sub-formula satisfies the model from the given state, the null update $\emptyset$ is returned. The heuristic would class this as the most minimal update set, as it would effect no modification to the underlying model and thus be vacuously closest to the original model. The modularity of the heuristic function allows it to be swapped out in the case a different set of minimal update semantics are required.

3.2.8 AX Formula Update

The function $Update_{AX}$ in Algorithm 3.7 is a derived implementation of the semantic characterisations in Subsection 2.4.2, from the previous chapter and is the base case for universal temporal operations. The semantics of universal next requires that all immediate successor states from the current state satisfy the sub-formula given. This was formally defined as $(M,\ s) \models \mathrm{AX}\phi$ iff $\forall s_1$ such that $(s,\ s_1) \in R, (M,\ s_1) \models \phi$ in Definition 2.3.

There are four cases which need to be taken into account and three possible atomic methods for update which will satisfy these cases. Firstly there exists the case where the current state is a leaf, having no outward relations (*line 7*). For this contingency there are two possibilities. We can create a new state s^* with the set of labels of the current state s and add a relation $r = (s, s^*)$ between the current and new state and apply the update procedure to the new state, using the sub-formula ϕ_0. This can be represented formally as $S' = S \cup \{s^*\}$, $R' = R \cup \{(s, s^*)\}$ and $L'(s^*) = L(s)$. In the algorithm, this is represented as the update tuple set

$\mathcal{U} = \{(s^*, +, Null),\ ((s,\ s^*), +, Null),\ (s^*, +, L(s))\} \cup Update_c(M, s^*, \phi_0)$. It is also possible to introduce a new self-loop to the relation $R' = R \cup \{(s, s)\}$ and update the state by the sub-formula, such that there exists a path in the model which satisfies universal next semantics.

Algorithm 3.7: $Update_{AX}(M,\ s,\ \phi)$.

Input: $M = (S,\ R,\ L),\ \phi,\ s$, where $s \in S$ and $(M,\ s) \not\models \phi$;

Output: Update set $\mathcal{U}$, where $u \in \mathcal{U}$ and $u = (s|r,\ +|-,\ p)$, where $s|r$ is the state or relation modified, p the related atomic proposition, and $+|-$ the update modifier.

01: $Update_{AX}(M,\ s,\ \phi)$
02: {
03: **if** $(s, s) \in R$ **and** $(M, s) \not\models \phi_0$: `//self loop`
04: 1. **if** $\exists x$, **where** $(s,\ x) \in R$, $(M, x) \models \phi_0$ **and** $x \neq s$:
05: $\mathcal{U} = \mathcal{U} \cup \{((s,\ s), -, Null)\}$;
06: 2. $\mathcal{U} = \mathcal{U} \cup Update_c(M, s, \phi_0)$;
07: **if** $\nexists x \in S$, **where** $\{(s,\ x) \in R\}$: `//no branches`
08: $\mathcal{U} = \mathcal{U} \cup \{(s^*, +, Null)\} \cup \{(s^*, +, L(s))\}$ $\cdots \cup Update_c(M,\ s^*,\ \phi_0)$;
09: **if** $\exists x \in S$, **where** $\{(s, x) \in R\} \wedge (M,\ x) \models \phi_0$: `//branch and there exists satisfied states`
10: **for some** $x \in S$ **where** $\{(s, x) \in R\}$ **and** $(M,\ x) \not\models \phi_0$:
11: 1. $\mathcal{U} = \mathcal{U} \cup Update_c(M, x, \phi_0)$;
12: 2. $\mathcal{U} = \mathcal{U} \cup \{((s, x), -, Null)\}$;
13: **if** $\nexists x \in S$ **where** $\{(s,\ x) \in R\}$ **and** $(M,\ x) \models \phi_0$: `//branch, no satisfying states`
14: **for some** $x \in S$, **where** $\{(s,\ x) \in R\}$ **and** $(M,\ x) \not\models \phi_0$:
15: $\mathcal{U} = \mathcal{U} \cup Update_c(M,\ x,\ \phi_0)$;
16: **for all** $x \in S$, **where** $\{(s,\ x) \in R\}$ **and** $(M,\ x) \not\models \phi_0$:
17: 1. $\mathcal{U} = \mathcal{U} \cup Update_c(M,\ x,\ \phi_0)$;
18: 2. $\mathcal{U} = \mathcal{U} \cup \{((s,\ x), -, Null)\}$;
19: **return** $\mathcal{U}$;
20: }

Secondly, there is the case where the current state has successors, but no one state satisfies the sub-formula (*line 13*). In this case, we can use some combination of relation removal and state substitution, such that there exists at least one state transitioned to by the current state which satisfies the sub-formula. For state removal we represent an updated relation set as $R' = R - \{(s,\ x) | x \in S\}$ and state set as

$S' = S - \{s \in S\}$. State substitution can be done in the same way as mentioned in the propositional update case. A variation on this is the case where transitioning states exist which satisfy the sub-formula (*line 9*). In this case we may remove all transitions to states not satisfying the sub-formula or apply the same approach as if there were no satisfying states in the same way as referenced earlier.

Finally, there exists the case where a relation exists between a state and itself (*line 3*). In the case that there exist other satisfying branches the self loop can be removed to satisfy the property, otherwise the state can be substituted based on the sub-formula to complete the update.

3.2.9 AG Formula Update

As mentioned in the semantic characterisations section of the previous chapter, the universal global temporal token requires that every state reachable from the current state satisfy the sub-formula given. This was expressed as $(M,\ s) \models \text{AG}\phi$ iff $\forall\pi = [s_0,\ s_1, \ldots](s_0 = s)$ and $\forall s_i \in \pi, (M,\ s_i) \models \phi$, in Definition 2.3. To do this, we need to traverse the tree-like model from the state given as argument, in a depth-wise manner. We then apply $Update_c$ to each state to determine if each state satisfies the sub-formula. In the following chapter we go further into the specifics of how depth-first search is implemented in this system, in this chapter we abstract out the traversal process and just refer to paths.

The base case for update is the initial state not satisfying the underlying formula (*line 4*). In this case we cannot perform some update which removes the initial state, as there needs to exist at least one state which satisfies the global property and removing the root state makes any paths from the initial inaccessible. In the case the initial state does not satisfy the initial property, we call $Update_c$ to of replace the state s_0 with s_0^*, using the method proposed in $Update_p$ (if the sub-formula of AG is atomic). Otherwise, $Update_c$ will route it to the proper update, be it a temporal or propositional formula. It is possible that the sub-formula contains nested temporal operations; this can create scenarios where the satisfaction of two or more states can be enacted by applying some modification to the one area, in effect solving two or more property violations with one fix. To identify duplicate updates which

Algorithm 3.8: $Update_{AG}(M,\ s,\ \phi)$.

Input: $M = (S, R, L),\ \phi,\ s$, where $s \in S$ and $(M,\ s) \not\models \phi$;

Output: Update set $\mathcal{U}$, where $u \in \mathcal{U}$ and $u = (s|r,\ +|-,\ p)$, $s|r$ is the state or relation modified, p the related atomic proposition, and $+|-$ the update modifier.

```
01: Update_AG(M, s, φ) :
02: {
03:     for all paths π = [s_0, ..., s_i, ...] in M:
04:         if s == s_0 :
05:             U = U ∪ Update_c(M, s_i, φ_0);
06:         if s_i ∈ π and (M, s_i) ⊭ φ_1:
07:             U = U ∪ Update_c(M, s_i, φ_0);
08:             if ∄x ∈ S where {(s, x) ∈ R} : //s_i is a leaf
09:                 U = U ∪ {((s_{i-1}, s_i), −)};
10:     if Update_c(Update_apply(M, s, U), s, φ) == ∅ :
11:     //Update tuples satisfies formula
12:         return U;
13:     else:
14:         return -1;
15: }
```

satisfy the property for different states, update possibilities are iterated over and duplicates are eliminated in the update sets. Analysing the set, we can see that some smaller set of updates will satisfy each individual state applicable to the AG temporal operation.

From here, states can be replaced or removed to satisfy the property (*lines 7, 8*). Another important update which can occur with AG involves state leaves which do not satisfying the property. Applying state removal to leaves which do not satisfying the property preserves tree-like model structure, persistence conditions, and can return a model which is closest to the original based on weak bisimulation ordering.

In Section 5.2 of Chapter 5, we investigate a case study using temporal properties containing AG tokens with nested AF tokens. These properties assert that in all states of a local model, if some property is true, then at all future states some property will become true. We propose an algorithm for AF in the following section.

Example 3.6. *Consider the model M given in Example 3.1. We wish to guarantee that $(M, s) \models AG(a \wedge b \rightarrow AFc)$. Applying equivalences we push negation to propositional atoms and derive the property $\phi \equiv AG(\neg a \vee \neg b \vee AFc)$. We pass the model and property to a model checker to find $(M, s) \not\models \phi$, and receive three counterexamples explaining the violation. These include $\pi_1 = [s_1,\ s_2,\ s_5,\ s_3,\ s_4,\ s_7, s_3,$ $\ldots]$, $\pi_2 = [s_1,\ s_5,\ s_6]$, and $\pi_3 = [s_1,\ s_5,\ s_3,\ s_4,\ s_7,\ s_3,\ \ldots]$.*

Using these counterexamples, we construct a tree-like local model $M = (S, R, L)$, where $S = \{s_1,\ s_2,\ s_3,\ s_4,\ s_7,\ s_5,\ s_6,\ s_{25},\ s_{23},\ s_{24},\ s_{27}\}$, $R = \{(s_1,\ s_2),\ (s_2,\ s_{25}),$ $(s_5,\ s_3)\ (s_3,\ s_4), (s_4,\ s_7),\ (s_7,\ s_3), (s_1,\ s_5), (s_5,\ s_6), (s_{25},\ s_{23}),\ (s_{23},\ s_{24}),$ $(s_{24},\ s_{27}),\ (s_{27},\ s_{23})\}$ and $L(s_1) = \{a\}$, $L(s_2) = L(s_3) = L(s_4) = L(s_6) = L(s_7) = L(s_{23}) = L(s_{24}) = L(s_{27}) = \{b\}$, $L(s_5) = L(s_{25}) = \{a,\ b\}$.

Applying $Update_c$ routes control to $Update_{AG}$ with state s_1 and property $\neg a \vee \neg b \vee AFc$. Applying propositional logic to s_1 finds $(M,\ s_1) \models \neg b$, such that its branches can be checked for satisfaction. Applying $Update_c$ to s_5 finds $(M,\ s_5) \not\models \neg(a \wedge b)$ and s_5 has no reachable future state which satisfies c. Thus, the modifications which will satisfy this branch of the local model are $\mathcal{U}_1 = \{(s_5, +, a)\}$, $\mathcal{U}_2 = \{(s_5, +, b)\}$, $\mathcal{U}_3 = \{(s_3, +, c),\ (s_6, +, c)\}$. Applying $minChange()$ finds that of the three candidates $\mathcal{U}_3$ is the most minimal.

Applying this process to the second branch of s_1 finds s_2 satisfies the sub-property of AG $(M, s_2 \models \neg a)$ but $(M, s_{25}) \not\models \neg a \vee \neg b \vee AFc$. Following the approach used for the former branch we see the updates possible include $\mathcal{U}_1 = \{(s_{25}, +, a)\}$, $\mathcal{U}_2 = \{(s_{25}, +, b)\}$, $\mathcal{U}_3 = \{(s_{23}, +, c)\}$. Applying $minChange()$ finds that of the three candidates $\mathcal{U}_3$ is the most minimal. On termination, $Update_{AG}$ returns the update tuple set $\mathcal{U} = \{(s_3, +, c),\ (s_6, +, c),\ (s_{23}, +, c)\}$.

3.2.10 AF Formula Update

To satisfy a model (M, s) by a universal future token, the sub-formula of the AF token needs to be satisfied at some individual future state, along all paths accessible from the current state. In Chapter 2, Definition 2.3 this was formally referred as $(M,\ s) \models \text{AF}\phi$ iff $\forall \pi = [s_0,\ s_1, \ldots](s_0 = s), \exists s_i \in \pi$, such that $(M,\ s_i) \models \phi$[5].

Algorithm 3.9: $Update_{AF}(M, s, \phi)$.

Input: $M = (S, R, L), \phi$, s where $s \in S$ and $(M,\ s) \not\models \phi$;

Output: Update set $\mathcal{U}$, where $u \in \mathcal{U}$ and $u = (s|r,\ +|-,\ p)$, where $s|r$ is the state or relation modified, p the related atomic proposition, and $+|-$ the update modifier.

01: $Update_{AF}(M, s, \phi)$:
02: {
03: **for all** paths $\pi_i = [s_0,\ \ldots,\ s_i,\ \ldots]$ in M:
04: **for** s_i in π_i:
05: $\exists s_i$ s.t. $(s_i,\ s_i) \in R \wedge M, s_i \not\models \phi_0$:
06: $\mathcal{U} = \mathcal{U} \cup Update_c(M, s_i, \phi_0)$; `//self loop case`
07: $\nexists x \in S$ s.t. $x \in \pi \wedge (M,\ x) \models \phi_0$:
08: select $s \in S$ s.t. $\nexists x \in S$ where $\{(s,\ x) \in R\}$:
09: $\mathcal{U} = \mathcal{U} \cup Update_c(M, s_i, \phi_0)$; `//leaf case`
10: $\exists \pi = [s_0,\ \ldots,\ s_{i-1},\ s_i,\ \ldots,\ s_i,\ \ldots]$: `//cyclic case`
11: select some $s_i \in \pi$ s.t. $(M,\ s_i) \not\models \phi_0$:
12: 1. $\mathcal{U} = \mathcal{U} \cup Update_c(M, s_i, \phi_0)$;
13: **for all** $s_x \in \pi = [s_0,\ \ldots,\ s_i]$ **where** $\exists s_y, s'_y$ s.t. $\cdots\ (s_y \neq s'_y, (s_x, s_y) \in R$ **and** $(s_x, s'_y) \in R)$:
14: $\mathcal{U} = \mathcal{U} \cup Update_{AF}(M, s_x, \phi_0)$;
15: 2. $\mathcal{U} = \mathcal{U} \cup Update_c(M, s_{i+1}, \phi_1)$;
16: **return** $\mathcal{U}$;
17: }

To obtain the most minimal update by weak bisimulation semantics, as illustrated in the semantic characterisations, update needs to occur at path leaves, state self loop relations and at some state within SCCs (*lines 5, 10 and 15*). Modification only needs to occur on all unsatisfied branches once; if a path is satisfied once

[5]In implementation, the equivalence of AFϕ with A[$\top$ U ϕ] is used to promote code re-use of $Update_{AU}$.

closer to the initial state, the associated lower paths are released from satisfying the sub-formula. As local models will have a tree-like structure, traversal can continue over paths which have yet to be satisfied. In implementation, reference to the states are saved on a stack and accessed when the current path is satisfied.

Another possibility for minimal update given previously to satisfy persistence properties, is to extend a finite path by creating a new state s^*, such that $S' = S \cup \{s^*\}$ and introducing a relation between s^* and an unsatisfied leaf state s on the path. The state's label function can be assigned the label function of s, such that $L(s^*) = L(s)$ and can be updated by $Update_c$ using the sub-formula ϕ. This constitutes the update set $\mathcal{U} = \{(s^*,\ +, Null),\ ((s,\ s^*),\ +, Null)\} \cup Update(M,\ s^*, \phi) \cup \{(s^*, +, L(s))\}$.

Strongly connected components (SCCs) in tree-like models are also a special case for consideration as illustrated in semantic characterisations. As the notion of tree depth does not apply for SCCs due to their cyclic nature, we need to consider which state to update in a SCC and apply accordingly. As a SCC constitutes an infinite path, some state needs to satisfy the property for the path. Two reasonable places to apply update would be the first state leading into the SCC from the path, or the final state of the SCC. We could then apply AF semantics to any subsequent state within the SCC which has linear paths transitioning outwards from the SCC that could constitute an unsatisfied path. In this implementation we consider the former.

Example 3.7. *Consider the model M represented by the transition graph in Figure 3.1. Suppose we require $(M,\ s)$ to satisfy property $AX(AF(b \rightarrow c))$. Passing M and ϕ to a model checker, we derive the counterexamples describing the paths $\pi_1 = [s_1,\ s_2,\ s_3,\ s_4,\ s_7,\ s_3,\ \ldots]$, $\pi_2 = [s_1,\ s_5,\ s_6]$ and $\pi_3 = [s_1,\ s_2,\ s_5,\ s_3,\ s_4,\ s_7,\ s_3,\ \ldots]$. Using these counterexamples we create the tree-like local model $C = (S,\ R,\ L)$ where $S = \{s_1,\ s_2,\ s_3,\ s_4,\ s_7,\ s_5,\ s_6,\ \ s_{25},\ s_{23},\ s_{24},\ s_{27}\}$, $R = \{(s_1,\ s_2),\ (s_2,\ s_3),\ (s_3,\ s_4),\ (s_4,\ s_7),\ (s_7,\ s_3), (s_1,\ s_5), (s_5,\ s_6), (s_2,\ s_{25}), (s_{25},\ s_{23}),\ (s_{23},\ s_{24}),\ (s_{24},\ s_{27})\ , (s_{27},\ s_{23})\}$ and $L(s_1) = \{a\}$, $L(s_2) = L(s_3) = L(s_4) = L(s_6) = L(s_7) = L(s_{23}) = L(s_{24}) = L(s_{27}) = \{b\}$, $L(s_5) = L(s_{25}) = \{a,\ b\}$.*

Applying equivalences to ϕ, we transform $AXAF(b \rightarrow c)$ into $AXAF(\neg b \vee c)$ and use $Update_c(M, s, \phi)$ to derive a modification. Control routes to $Update_{AX}$ which then applies $Update_{AF}$ to s_5 and s_2 by formula $\neg b \vee c$. To satisfy the formula at these states some future state needs to satisfy $\neg b \vee c$, such that $s \not\models b$ or $s \models c$. Applying $Update_{AF}$ to s_5 yields s_6 being updated by c, based on s_6 being a leaf state. This returns the update tuple $(s_6, +, c)$. On the next branch from s_2, $Update_{AF}$ returns the update tuples $(s_3, +, c)$ and $(s_{23}, +, c)$. This gives the overall returned update as $\mathcal{U} = \{(s_6, +, c),\ (s_3, +, c),\ (s_{23}, +, c)\}$.

3.2.11 AU Formula Update

Semantics of AU requires that the first property of the sub-formula hold along all paths accessible from the current state, until the second property argument is satisfied on the given path. In Definition 2.3 of the prior chapter, we formally defined this as $(M, s) \models \mathrm{A}[\phi_1\ \mathrm{U}\ \phi_2]$ iff $\forall \pi = [s_0,\ s_1, \ldots](s_0 = s), \exists s_i \in \pi$, such that $(M,\ s_i) \models \phi_2$ and $\forall j < i, (M,\ s_j) \models \phi_1$. For the purpose of identification, the first property will be referred to as the pre-condition and the second the post-condition. Each path in this case can be vacuously satisfied by making the initial state s_0 satisfy the post-condition, or traversing each path until a state is encountered which satisfies neither property and updating it by the post-condition sub-formula. While the first is not a minimal update, the second will constitute a minimal update, as it is necessary to have the state satisfy the property by one of the sub-properties. To enact the most minimal change, each tree-like path is traversed in a depth-wise manner, much like in $Update_{AF}$.

Each state along the path is checked to determine if it satisfies the pre-condition and post-condition sub-formulas ϕ_1 and ϕ_2. If it satisfies the post-condition, the branch is satisfied and a new branch previously unchecked can be analysed (*line 4*). If the state satisfies the pre-condition sub-formula and it is a leaf state, a self looping state $(s,\ s) \in R$, or SCC entry state, the state is updated by the post-condition formula and an unchecked branch is analysed (*line 6*). If the state satisfies the pre-condition and doesn't satisfy the earlier conditions, we move to the next available state at a lower tree-branch and apply the AU process. Otherwise if a state satisfies neither the pre-condition or post-condition formula, $Update_c$ can be called to satisfy

Algorithm 3.10: $Update_{AU}(M,\ s,\ \phi)$.

Input: $M = (S, R, L)$, ϕ, s, where $s \in S$ and $(M,\ s) \not\models \phi$;

Output: Update set $\mathcal{U}$, where $u \in \mathcal{U}$ and $u = (s|r,\ +|-,\ p)$, where $s|r$ is the state or relation modified, p the related atomic proposition, and $+|-$ the update modifier.

01: $Update_{AU}(M,\ s,\ \phi)$:
02: {
03: **for all** paths $\pi = [s_0,\ \ldots,\ s_i,\ \ldots]$ in M:
04: select least $s_i \in \pi$ where $(M,\ s) \not\models \phi_0 \vee \phi_1$:
05: $\mathcal{U} = \mathcal{U} \cup Update_c(M, s_i, \phi_1)$;
`//unsatisfied path where neither formula is true`
06: $\exists s_i$ s.t. $(s_i,\ s_i) \in R$ **and** $(M, s_i) \not\models \phi_1$:
07: $\mathcal{U} = \mathcal{U} \cup Update_c(M, s_i, \phi_1)$; `//self loop case`
08: $\exists s_i$ s.t. $s_i \in \pi \wedge \nexists x$ where $\{(s,\ x) \in R\} \wedge (M, s_i) \not\models \phi_0$:
`//leaf case`
09: 1. $\mathcal{U} = \mathcal{U} \cup Update_c(M, s_i, \phi_1)$;
10: 2. $\mathcal{U} = \mathcal{U} \cup \{(s^*, +, \phi_1),\ ((s,\ s^*),\ +)\}$
$\cup\ Update_c(M,\ s^*,\ \phi_1)$;
11: $\exists \pi = [s_0,\ \ldots,\ s_{i-1},\ s_i,\ \ldots,\ s_i,\ \ldots]$: `//cyclic case`
12: select some $s_i \in \pi$ such $(M,\ s_i) \not\models \phi_1$:
13: **if** $(M,\ s_i) \not\models \phi_0$:
14: $\mathcal{U} = \mathcal{U} \cup \{Update_c(M, s_i, \phi_1)\}$;
15: **for all** $s_x \in \pi = [s_0,\ \ldots,\ s_i]$ **where** $\exists s_y, s'_y$ s.t.
$\cdots\ (s_y \neq s'_y, (s_x, s_y) \in R$ **and** $(s_x, s'_y) \in R)$:
16: $\mathcal{U} = \mathcal{U} \cup Update_{AU}(M, s_y, \phi)$;
17: **return** $\mathcal{U}$;
18: }

the branch based on the post-condition sub-formula. It is also possible to extend the path by creating a new state with the previous states label function, but updated by the post-condition and a relation occurring between the leaf state and new state. This update rests on the condition that the path leading from the initial state satisfies the pre-condition at each state but does not satisfy the post-condition in the path.

As with AF, path semantics for AU require that the post-condition be satisfied at some state in a SCC, AU differs in that the post-condition satisfying state be trailed by states satisfying the pre-condition. To demonstrate a case of a local model being

updated by an ACTL formula containing an AU token, we give Example 3.7.

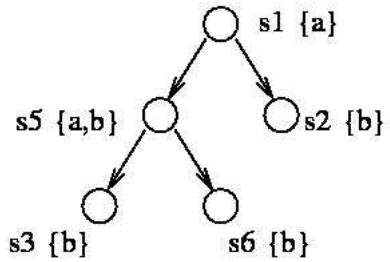


Figure 3.5: Tree-like model generated from counterexamples of $(M, s) \not\models$ A[a U c] in Example 3.5.

Example 3.8. *Consider the Kripke model represented in the transition graph in Figure 3.1. Suppose we require the model* $(M,\ s_1)$ *satisfy the properties* $A[a\ U\ AG(b)]$ *and* $A[a\ U\ c]$*. Applying this to the model, we find that although it satisfies the prior property, the latter is unsatisfied, generating the counterexamples* $\pi_1 = [s_1,\ s_2]$, $\pi_2 = [s_1,\ s_5,\ s_2]$, $\pi_3 = [s_1,\ s_5,\ s_6]$*. We use these counterexamples to generate a tree-like model* $M = (S, R, L)$*, where* $S = \{s_1,\ s_2,\ s_3,\ s_5,\ s_6\}$, $T = \{(s_1,\ s_2),\ (s_1,\ s_5),\ (s_5,\ s_3),\ (s_5,\ s_6)\}$, $L(s_1) = \{a\}$, $L(s_2) = L(s_3) = L(s_6) = \{b\}$, $L(s_5) = \{a,\ b\}$*, shown in Figure 3.5.*

We apply the update function $Update_c$ *with* M, s_1 *and the property* $\phi \equiv A[a\ U\ c]$*.* $Update_c$ *passes control to* $Update_{AU}$ *with the former arguments and* $Update_{AU}$ *ascertains satisfaction of a and b individually at* s_1*, through* $Update_p$*.* $Update_p$ *finds* s_1 *satisfies the pre-condition a but not c; with this, the* $Update_{AU}$ *picks the branch with* s_5 *and saves the other branch on a stack for processing.* $Update_{AU}$ *applies the same process to* s_5*, finding it also satisfies the pre-condition, again passing to the lower branch* s_3 *and saving* s_6 *on the stack for branches to visit.* s_3 *is found by* $Update_p$ *to satisfy neither a or c. To solve this, the most minimal change would be to replace* s_3 *with a state that satisfies its previous labels, but also c. This is represented in the update set* $\mathcal{U} = \{(s_3, +, c)\}$*. As this was a leaf state and now satisfies the formula, the next branch is checked from the branch stack,* s_6*.* s_6 *is found to also be a leaf state and can be handled in the same manner* s_3 *was, giving a running update set of* $\mathcal{U} = \{(s_3, +, c),\ (s_6, +, c)\}$*. It is also the same for the final branch on the stack* s_2*, which satisfies neither condition. Being the final branch, the algorithm terminates, returning the update set of* $\mathcal{U} = \{(s_3, +, c),\ (s_6, +, c)\}$.

3.3 Constraint Automata

To enact constraint compliance as described in Definition 2.10, we define two separate functions with the tasks of modifying a tree-like model to comply to the variable and action constraints given for a model update session. To begin we analyse the variable constraint automata compliance algorithm *ComplianceV*.

ComplianceV first examines each transition function $\delta(s_p,\ a) = s_q$ in $\mathcal{VC}(V,\ A)$, from which it identifies those state transitions $(s_i,\ s_j)$ in model $(M,\ s)$ not complying to such variable bindings through action a. Then *ComplianceV* will minimally change the corresponding states so that the newly formed model complies to the given variable constraint automaton. Note that in *ComplianceV*, s_p, s_q are states in automaton $\mathcal{VC}(V,\ A)$, s_i, s_j are states in model $(M,\ s)$, and s'_j is a state in model $(M',\ s')$. This is achieved in such a way that *ComplianceV* is recursively calling itself (*i.e. line 8*).

Algorithm 3.11: $ComplianceV((M,\ s),\ \mathcal{VC}(V,\ A))$.

Input: A tree-like model $(M,\ s)$ and a variable constraint automaton $\mathcal{VC}(V, A)$;
Output: Tree-like model (M', s') complying to $\mathcal{VC}(V, A)$;

```
01: ComplianceV((M, s), VC(V, A)) :
02: {
03:     for all (s_i, s_j) in M labelled by action a:
04:         if there exists a transition δ(s_p, a) = s_q in VC(V, A)
05:         and s_p ⊆ s_i:
06:             if s_q ⊈ s_j:
07:                 form a new tree-like model (M', s') in which s_j
                        is replaced by s'_j such that s_q ⊆ s'_j and
                        Diff(L(s_j), L(s'_j)) is minimal;
08:                 ComplianceV((M', s'), VC(V, A));
09:         else:
10:             return (M', s');
11: }
```

ComplianceA works in a similar fashion to *ComplianceV*; it examines each transition function $\delta(s^a_p,\ a) = s^a_q$ in $\mathcal{AC}(A)$ and each state s^m_i in S, then seeks to match

each s_q^a to some s_j^m in $\mathcal{AC}(A)$ (*lines 3, 7 and 11*). We can then determine if there is some s_j^m which corresponds to some s_q^a, violating the semantics of the given action a, described in Definition 2.9. In *ComplianceA* s_p^a, s_q^a refer to states in the automata and s_i^m, s_j^m refer to states within the model. *ComplianceA* finishes by recursively calling itself to guarantee other branches comply to the automaton and the modification effected compliance.

Algorithm 3.12: *ComplianceA*$((M,\ s),\ \mathcal{AC}(A))$.

Input: A tree-like model $(M,\ s)$ and a variable constraint automaton $\mathcal{AC}(A)$;
Output: Tree-like model (M', s') complying to $\mathcal{AC}(A)$;

01: *ComplianceA*$((M, s), \mathcal{AC}(A))$:
02: {
03: **if** for some $L(s_i) \subseteq L(s_a)$ and some s_j, $\delta(s_i,\ next) = s_j$:
04: **if** there exists some $(s_a, s_b) \in R$ in M and $s_j \subseteq s_b$
... does not hold:
05: form a new tree-like model (M', s') in which all s_b are replaced by s'_b such that $L(s_j) \subseteq L(s'_b)$ and $\mathsf{Diff}(L(s_j), L(s'_b))$ is minimal;
06: **if** for some $L(s_i) \subseteq L(s_a)$ and some s_j, $\delta(s_i,\ preceded) = s_j$:
07: **if** there exists some path $\pi = [s_0, \ldots, s_a, s_{a+1}, \ldots, s_b, s_{b+1}, \ldots]$ in M where $L(s_i) \subseteq L(s_a)$ and no s_b where $L(s_j) \subseteq L(s_b)$ and $s_b > s_a$.
08: form a new tree-like model (M', s') in which there is some s_b where $s_b > s_a$ is replaced by s'_b such that $L(s_j) \subseteq L(s'_b)$ and $\mathsf{Diff}(L(s_b), L(s'_b))$ is minimal;
09: **if** for some $L(s_i) \subseteq L(s_a)$ and some s_j, $\delta(s_i,\ exclusive) = s_j$:
10: **if** there exists some path $\pi = [s_0, \ldots, s_a, s_{a+1}, \ldots, s_b, s_{b+1}, \ldots]$ ... where $L(s_j) \subseteq L(s_b)$ in M and $s_j \subseteq s_b$ where $L(s_i) \subseteq L(s_a)$ and there is some s_b
11: form a new tree-like model (M', s') in which s_b is replaced by s'_b such that $L(s_j) \not\subseteq L(s'_b)$ and $\mathsf{Diff}(L(s_b), L(s'_b))$ is minimal;
12: *ComplianceA*$((M',\ s'),\ \mathcal{AC}(A))$;
13: **else**:
14: **return** (M', s').
15: }

3.4 Algorithm Example - Microwave Oven

To demonstrate the application of algorithm we will apply the approach to the microwave oven model example from Chapter 1. As shown earlier, the microwave oven model is a simple example for demonstrating how seemingly correct approaches can violate common-sense safety properties. Earlier we extracted two counterexamples which explained the violation in the model of the property $\phi \equiv \text{AG}(start \rightarrow \text{AF}(heat))$. One of the counterexamples contains a SCC witnessing *Start* satisfied at a state and $Heat$ never becoming true at any state in the SCC. This describes the infinite path $\pi_1 = [s_1,\ s_2,\ s_5,\ s_3,\ \ldots,\ s_1,\ \ldots\ ]$ and can be described with the local model $M = (S,\ R,\ L)$, where $S = \{s_1,\ s_2,\ s_3,\ s_4\}$, $R = \{(s_1,\ s_2),\ (s_2,\ s_5),\ (s_5,\ s_3),\ (s_3,\ s_1)\}$, and $L(s_1) = \emptyset$, $L(s_2) = \{Start,\ Error\}$, $L(s_5) = \{Start,\ Close,\ Error\}$, $L(s_3) = \{Close\}$. To begin the process we execute the main function $Update_c$, passing the local model M, initial state s_1 and the property ϕ.

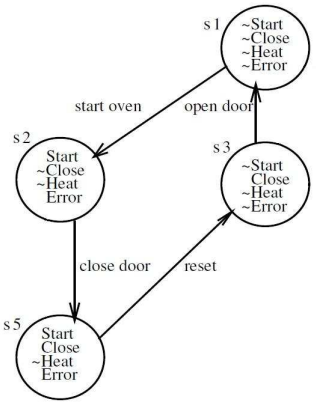


Figure 3.6: Counterexample for $(M,\ s) \not\models \text{AG}(start \rightarrow \text{AF}(heat))$.

The property is analysed to determine which function to route control to based on property semantics. As AG is the most immediate in the formula parse tree $Update_{AG}$ is called with the model, initial state and sub-formula $start \rightarrow \text{AF}(heat)$ as argument. Semantics of AG dictate that each state accessible from the current state should satisfy the sub-property. We traverse the sub-model from s_1, creating a new traversal object that successively returns states in the SCC, and then calling

$Update_c$ on each state with the sub-property. Applying the sub-property to s_1 we find that we need to ascertain if $\neg start$ is true or if at some given future *heat* becomes true ($Update_c$ routes control to $Update_\vee$). Here s_1 is found to satisfy $\neg start$ as $start \notin L(s_1)$. This will return the null update $\emptyset$ and $Update_\vee$ will also return the null update as one of the two terms are satisfied, thus satisfying semantics. $Update_{AG}$ will find s_1 satisfies the sub-property.

Checking s_2 will find it does not satisfy the property, as $Update_\vee$ and subsequently $Update_p$ will be called for $\neg start$; $Update_p$ will find $start \notin L(s_2)$. The system will return the possible repair of $u = (s_2,\ -,\ Start)$. Returning to $Update_\vee$ its second argument AF($heat$) is checked. From s_2 each path is checked to determine if any state satisfies the sub-formula *heat*. As there is only one path, a SCC, we check every state and find no state on the path satisfies *heat*. As notions of depth do not apply to cyclic behavior we may update at the first state in the SCC $Heat$. This will satisfy s_2, we also need to consider s_5 and s_3.

Applying the same process to s_5 and s_3 we find neither satisfies the property and both may be modified to satisfy the property by updating the state by "$\neg start$" or by updating some arbitrary state in the future by $Heat$. To satisfy AG temporal operator semantics each state must satisfy the property. Combining the possible updates for each individual state we can substitute each state by a new state satisfying $\neg start$, substitute some state in the SCC by a new state satisfying *heat* once to satisfy all states, or some combination of the two. We can see that the approach which is most minimal based on weak bimsimulation semantics in this case is to update one arbitrary state once by *heat* in the SCC, s_2'. This is because this one update will satisfy both s_5 and s_2 with one update, whereas negating *start* requires substitution of both s_5 and s_2. The model can then be re-checked to guarantee the satisfaction of the property still holds for the model as a whole. Finally we derive a model $M' = (S',\ R',\ L')$ using the update set $\mathcal{U}$, where $S' = \{s_1,\ s_2',\ s_5,\ s_3\}$, $R' = \{(s_1, s_2'),\ (s_2', s_5),\ (s_5, s_3),\ (s_3, s_1)\}$, $L(s_5') = \{start,\ heat,\ error\}$ as shown in Figure 3.7.

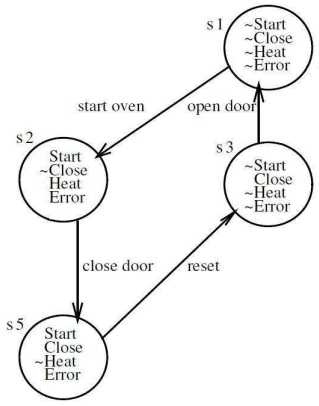


Figure 3.7: Update $(M',\ s) \models \mathrm{AG}(start \rightarrow \mathrm{AF}(heat))$.

3.5 Summary

The algorithms presented in this chapter are an abstract framework and represent the concepts of localised update embodied in algorithms and were originally derived from the characterisations. In the following section we will go into the implementation details of the prototype local model update system, discussing formula parsing, highlight important elements of the core functionality and show how the algorithms are utilised to effect update in the NuSMV language specific context. This will give a more grounded and detailed view of the system implementation and more insight into application.

Chapter 4

Local Model Update Prototype - *l-Up*

4.1 Introduction

In earlier chapters we have explained how we generate abstract algorithms for enacting localised model update over tree-like models. In this chapter we describe the capabilities of the *l-Up* local model update prototype software tool, and examine the subtleties and limitations of the prototype generated. *l-Up* was designed in the Python programming language with an emphasis of readability and future adaption by researchers in this field. Python was also adopted for its reputations for rapid prototyping and sound code in industry and the AI community. Rapid prototyping was a desired attribute as the PhD project has a restricted time frame; algorithms in this chapter are simplified versions of the coded implementation designed in the Python programming language to assist readability. The full implementation can be accessed from the School of Computing and Mathematics page

http://scm.uws.edu.au/∼mkelly/l-upProject.html

4.1.1 Restrictions on Prototype Development

For the sake of clarity, we stress that this initial implementation should be taken as a prototype to demonstrate a proof of concept for the theory presented in this dissertation. At this current point the prototype can only provide candidate fixes

for tree-like models in two forms, simple Kripke structures as demonstrated in the examples presented in previous chapters, or through local models generated from counterexamples extracted from NuSMV model checking sessions. The local models generated from NuSMV sessions are limited to single finite or infinite counterexample traces. This is because NuSMV only returns the single counterexample for a given temporal property in a checking session. The capability, however, exists in *l-Up* for handling tree-like local models.

As shown in the characterisations of Chapter 2, complexity limits the size of local models and nesting depth of temporal properties passed to the prototype. We will demonstrate this further in case studies presented in the following chapter and show that time to compute grows quickly with an increase of model size, and with nesting of temporal properties. This prototype was generated as a means of demonstrating the feasibility of update localisation to isolate a fault location and reduce the repair space to the region causing the fault. The system will provide candidate fixes and extract information from the SMV model specification file including checking session details to assist the developer.

In the future we hope to see this prototype extended such that it can be used as a global modification to a NuSMV specified model based on temporal specifications. At the current time the following features are available:

- Extracts variables and module information from a given smv file to assist update;
- Translates the counterexamples into interpretable local models;
- Computes candidate local model updates such that the local model satisfies a given temporal property;
- Computes constraints such that constraint compliance is upheld.

With these defined features, we will comment on how the prototype can be extended and implemented as a full universal local model update system in the conclusion of this dissertation.

4.1.2 System Overview

The system presented in this thesis uses a layered hierarchy to handle the necessary levels of complexity in generating a candidate fix on a local model. The central script *session.py* provides an interface to generate an update at a session level and provides access to the majority of functionality provided in the other packages developed for the project. From this script, the user can specify a model file location, temporal properties, a constraint file and any debugging flags required.

In Figure 4.1, we show a high-level representation of the *l-Up* prototype structure. Further to the functionality discussed in the previous section we also note the presence of the model checker NuSMV and the files used to specify the model and constraint automata. Here, NuSMV has the effect of determining the satisfaction of the temporal formula in the model specification file and returning counterexamples explaining the failure in the model to satisfy the formula.

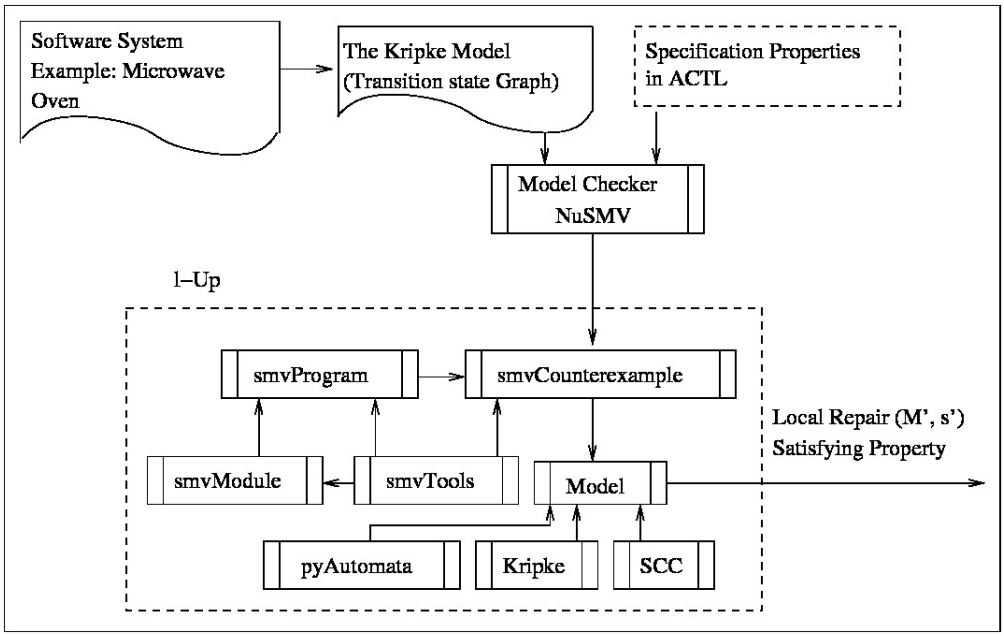


Figure 4.1: Implementation graph.

Being a prototype, the files used for the update process are saved and all details to an update session are stored in a reporting folder for analysis and such that the system can be built upon and better researched in future development and debugging. Another important note is that *l-Up* requires the *PLY 3.3* package available at *http://www.dabeaz.com/ply/*. This is necessary for the *yacc* and *lex* modules to implement the parsers for the *smvModule* module, which parses macro variable definitions, and also the *formulaeOperators* package prototype parser to operate.

4.1.3 Prototype Development

Software Architecture and Technical Specifications

l-Up is written using the scripting language Python (version 2.6.2) [50] in the PyDev project development environment (version 1.5.0.1) [2] and the software application platform *Eclipse* (version 3.5.1, build M20090917-0800) [51]. In the following section we will discuss important functionality used for providing update and interfacing with NuSMV and comment on design aspects employed. For each Python module, we will provide algorithms expressed as pseudo-code to exhibit the functionality offered through the classes of each module.

In Figure 4.2 we show the file hierarchy for the *l-Up* project. On the shallowest level (*lines 1, 9, 12 and 26*) we have the main packages providing the modules for the update suite and the session module (*line 31*) which brings together each of the packages and provides an abstraction layer for the developer to interact with the aspects of the update system at a high-level.

In this chapter we will demonstrate many of the features of *l-Up* SMV model data extraction. We will use an example file available online of the *gigamax cache coherence protocol* originally given in Ken MacMillans' thesis *Symbolic Model Checking* [75] and also available from the NuSMV example repository [100][1]. *gigamax.smv* was selected as a program for parsing for its clear syntax. It utilises many aspects of NuSMV syntax and is non-trivial in terms of program size and modularity.

[1] See appendix A for referenced programs.

```
l-Up Project
01:  ├── pyAutomata            #Constraint Modules
02:  │    ├ action.py
03:  │    ├ actionAutomata.py
04:  │    ├ automataHandler.py
05:  │    ├ varAutomata.py
06:  │    ├ variableConstraintCheck.py
07:  │    ├ violation.py
08:  ├── pyModel               #Local Model Update Modules
09:  │    ├ atom.py
10:  │    ├ auxFunctions.py
11:  │    ├ CEXReduct.py
12:  │    ├ elements.py
13:  │    ├ kripke.py
14:  │    ├ metrics.py
15:  │    ├ model.py
16:  │    ├ modelTraverser.py
17:  │    ├ scc.py
18:  │    ├ treelike.py
19:  │    ├ updateHeuristics.py
20:  │    ├ updateType.py
21:  ├── pySMV                 #Modules for parsing SMV content
22:  │    ├ smvCounterexample.py
23:  │    ├ smvModule.py
24:  │    ├ smvProgram.py
25:  │    ├ smvTools.py
26:  │    ├ smvTransition.py
27:  ├── pyFormula
28:  │    ├ ACTLOps.py
29:  │    ├ ACTLFormulaParser.py
30:  │    ├ evalOperators.py
31:  ├── session.py             #Script to initiate an update session
```

Figure 4.2: File hierarchy for the *l-Up* local model update project.

Model Checking Functionality

The local model updater described in this chapter works by deriving counterexamples from model checking sessions. To derive these counterexamples, we pass model specification files to the NuSMV model checker. For this project NuSMV 2.4.3 is used [16]. NuSMV is accessed via Python operating system calls with model files passed as argument. NuSMV is called using a non-interactive session and an output file of the result of the checking session is generated, labelled with a counterexample

identifier and the current timestamp to identify when it was created. Further, the flag $-r$ is used to indicate a verbose output, giving additional model details such as model state count, reachable states and model diameter.

Two levels of model checking are possible in the software presented: *l-Up* allows control to be passed to NuSMV for SMV program model checking. Also, *l-Up* provides a basic level of model checking at a prototype level with the simple programs given in the thesis theoretical section. This will be analysed in the model section of this chapter.

To initiate a NuSMV session, we pass a given program location to the NuSMV executable and indicate a file to output the session to. This is done with the system call

os.system("NuSMV\\2.4.3\\bin\\nusmv -r \\" + self.filePath + "\ >" ...
... +self.filePath.split("\\")[-1].replace(".smv", ' ') + "CounterEx.dat")

4.2 pySMV Package

Functionality of the local model update approach requires that the implementation has programmatic interaction with the NuSMV verification suite. This includes extracting information from both the model specification file and any subsequent counterexamples generated while checking a property. Both of these cases require a full understanding of NuSMV syntax and methods of storing this information for use in the prototype (and for future expansion to an industrial scope application). For this purpose we have included a package to handle this task, the *pySMV* for *l-Up* which includes the modules *smvCounterexample.py* for counterexample extraction, *smvProgram.py* for model specification file parsing, and *smvModule.py* for NuSMV file parsing at SMV program module level. The main Python modules of the *pySMV* package contains 1100 lines of Python code.

To begin this section we look at the ability of *l-Up* to extract domain dependent information from the given model file, including the *smvTools*, *smvModule* and *smvProgram* modules.

Assumptions

Counterexamples are derived from the model checking session and prior to this syntax of the NuSMV program is checked. If syntax errors are encountered, NuSMV will return and identify the found faults. Saying this, little error checking is necessary in extracting variable information from NuSMV program files. If errors are found or the program satisfies the temporal property NuSMV will return the appropriate signals and terminate.

The *pySMV* package is designed with extracting variable domain information from program specifications in mind. For the scope of this project information about transition relations and initial values are not taken from the program. Only information which allows us to determine variable domains for a local model is extracted in the package. This package can easily be extended at a later date to extract transition information; the *smvTools* package contains many regular expressions to handle this task. For the purposes of this prototype, the *pySMV* package can handle limited NuSMV syntax and type handling.

smvTools Module

In the package *pySMV*, modules share many common functionalities. To allow code reuse, the module *smvTools.py* provides common functionality to extract counterexample and program information. This includes storage of regular expressions for parsing NuSMV files, functions for creating a file pointer to NuSMV files for file I/O, extraction of file sections based on headers, and other miscellaneous functionality. Regular expressions in *smvTools* allow identification of reserved words from NuSMV files. This includes program specification reserved words, variable type definitions and counterexample syntax. Regular expressions referenced in this chapter for program and counterexample parsing are kept in *smvTools* and can be accessed by importing the module.

4.2.1 smvModule Module

Building a characterisation of each module in the smvProgram is an important step towards understanding the program as a whole. For this reason we have developed a class *smvModule* in the module *smvModule.py*, shown in Figure 4.1[2] with the task of identifying the elements of SMV modules such as arguments, defined variables, their types and domains and any relationships between declared modules or variables. With this information we could develop a hash table of SMV modules indexed by their hashable label, such that required information be derived by referencing the name and the module attribute we wish to query.

We extract much of this behaviour into a class object for a dual purpose. Firstly, this allows us to determine the nature of a variable given a valuation on a counterexample state and extract information pertaining to variable types and domains which will be useful when performing local model update. To be useful, we need to be able to trace variables back through the modules they are defined in and ascertain their type and domain. Secondly, this framework gives a foundation for automatically generating variable dependency graphs and other domain dependant information. This would allow us to automatically generate variable and action constraint automata to give improved update guidance and give future researchers a better platform from which to interact with NuSMV model files.

To begin with, the SMV module *main* is considered as a special module and is a reserved name for modules. Each NuSMV program will contain a module *main* from where other modules can be referenced as composite types. This allows more expressive behaviour in the constructed model. If a valid SMV program is analysed there will at least be one processed module structure, the *main* module. SMV modules can take arguments in the module header which effect the valuation of other variables and their transition relations. These work as variables within the scope of the referenced module.

To define the overall models transition system, modules define the initial and

[2]In the figure many of the methods have been removed for brevity. The smvModule module provides 30 methods as of writing of the dissertation, however only central methods are present.

transition values for variables locally using the *init()* and *next()* calls (discussed in Chapter 1). Modules also allow the declaration of additional attributes such as inheritance from other modules using the *is-a* relation to enrich module behaviour. We will analyse how we take into account these behaviours when extracting domain information for counterexamples.

Algorithm 4.1: smvModule class definition.

```
01: class smvModule():
02:     def __init__ (self, rawLines, debug = 0):
03:         self.moduleName = ""# string
04:         self.moduleArgs, self.isa = [], [] # lists
05:         self.variables, self.varTypes = {}, {}# hash tables
06:         self.defines = {}
07:     def buildModule(self, lines, debug = 0)
08:     def processVariables(self, lines)
09:     def processVariableString(self, varLine)
```

Variable Reference Hash Tables

Each variable of a program can be mapped back to the module where it was defined. For this reason we associate each module class object with hash tables indexing the domain and type of each variable by its label. To derive these variables the *smvModule* class object takes as argument a list of strings representing the variable declaration for the module (Raw file input represented as a list of strings beginning with the token VAR, and terminated by another SMV header, or the *end-of-file* sentinel value). These lines are iterated over and parsed using regular expressions defined in the *smvTools* module of the *smv* package. Variable types and domains are identified and stored in the declared *varType* and *variables* table, respectfully (*lines 6 and 7*).

Later in this section, we will look at reduction of variables in counterexamples into symbolic valuations at a state, such that variables are representable as propositional atoms in local models.

Comment:	'- -[\W\w]',
Variable Declaration Header:	'VAR[\W\w]*',
Temporal Specification:	'SPEC[\W\w]*',
Initial declarations:	'INIT[\W\w]*',
Variable next definition:	'next(\W\w)*',
Case Block:	'[\W\w]*case[\W\w]*',
End-case Block:	'[\W\w]*esac[\W\w]*',
Transition Assignment:	'([\W\w]*):([\W\w]*)',
Defines Assignment:	'([\W\w]*) := ([\W\w]*)',
Module Header:	'MODULE([\W\w]*)',
Assignment Header:	'ASSIGN[\W\w]*',
Defined variables:	'DEFINE[\W\w]*',
Initial Value for variable:	'init\(([\W\w])\([\W]:=([\W\w]);',
Assignment:	'([\W\w]*):[(]([\W\w]*)[)]',
Assignment:	'([\W\w]*);',
IS-A Declaration:	'ISA ([\W\w]*)',

Figure 4.3: Regular expressions for module data extraction.

Defined Variables

NuSMV allows a system for simplifying variables by declaring macro variables representing propositional formulas made up of other variables defined in a module. One example of this is in the *gigamax.smv* program in the *bus-device* module. *bus-device* defines an abort variable be true if the *REPLY-STALL* bit is set or *cmd* is set to *read-shared* or *read-owned* and the *REPLY-WAITING* bit is set. This is defined as:

$$abort := \text{REPLY-STALL} \mid (\text{CMD} = \text{read-shared} \mid \text{CMD} = \text{read-owned}) \\ \& \text{ REPLY-WAITING}.$$

These defined macro variables are used to the effect of succinctly defining common conditions which would otherwise need to be written in full in multiple parts of the module. Defined variables also give more information about their purpose in their defined label and can be used in temporal specifications as a means of syntactic sugaring. For *l-Up* to interact with NuSMV programs in any meaningful way, processing these defined variables is necessary. *l-Up* uses the formulae operators parser from PLY to parse these formulas and applies a nested structuring based on the formula tokens. These variable types are maintained in the *defines* hash table.

Module Inheritance in NuSMV

Another aspect of NuSMV modules is that defined modules can inherit behaviour from other modules. This has multiple purposes; chiefly it is a means of allowing code reuse. In *l-Up is-a* relationships can be interpreted and allow a module to reference variables defined in other modules. This is necessary to *l-Up*, when building a local model from SMV specification files and a counterexample trace *is-a* relationships allow reference to behaviour in other modules. Modules can then be checked to define a given variable, and if it does, the domain and type information for the referenced variable can be referenced in the local model. To show what is available through the *smvModule* class we give an example using the module in the *gigamax* class, *memory*.

Example 4.1. *A counterexample state makes reference to variables originating from the memory module of gigamax.smv. Calling the smvProgram class and instantiating an object smvProg with the gigamax.smv file name (using the Python object declaration smvProg = smvProgram(file_path + "gigamax.smv")), we reference the modules attribute of smvProg and use the label of memory to access the information for the specific module (memModule = smvProg.modules["memory"]). This passes the reference of the smvModule object to the variable memModule which has a module name available through memModule.moduleName. By referencing memModule.moduleArgs we find memory takes the arguments ["CMD", "REPLY-OWNED", "REPLY-WAITING", "REPLY-STALL"]. By referencing memModule.varTypes we find memory has 4 variables, 3 boolean (busy, master, and reply-stall) and an enumerable type variable cmd. To find the domain of the enumerable type variable cmd we reference memModule.variables["cmd"], which returns the list ["idle", "read-shared", "read-owned", "write-invalid", "write-shared", "write-resp-invalid", "write-resp-shared", "invalidate", "response"]. Finally, we find that the module "memory" defines a macro variable "abort" which is true if*

$$\begin{aligned} &\textit{REPLY-STALL} \mid (\textit{(CMD} = \textit{read-shared} \mid \textit{CMD} = \textit{read-owned)} \\ &\quad \& \textit{ REPLY-WAITING)} \mid \textit{((CMD} = \textit{read-shared} \\ &\quad \mid \textit{CMD} = \textit{read-owned)} \ \& \ \textit{REPLY-OWNED) is true}^{3}. \end{aligned}$$

[3]This formula structure is stored as a nested list such that its contents can be parsed recursively.

4.2.2 smvProgram Module

Having a method for representing each NuSMV module, we can analyse programs as a whole and get a global image of how variables interact between modules. By maintaining a hash table of modules indexed by their label, we can find process variable declarations to determine variable domains and types to aid in the construction of local models from counterexamples. The designed Python module *smvProgram* defines methods which use recursive back-tracing between modules to find what variable belongs to which module.

Algorithm 4.2: smvProgram class definition.

```
01: class smvProgram():
02:     def __init__ (self, filePath, debug = 0):
03:         self.filename = filePath.split("\\")[−1]
04:         self.modules = {}    # hash table
05:         self.stateCount = 0
06:     def lookupVariable(self, variable, module)
07:     def calculateModStateSpace(self, debug = 0)
08:     def recModules(self, modStateSpaceDict, module, debug = 0)
09:     def traceVariable(self, variableTraceList, moduleName = "main")
```

Variable Context Analysis

One of the challenges in extracting information from a NuSMV model file is determining which variable is defined in what module and linking those definitions to defined variables in other modules, such as the *main*. In NuSMV, *attribute dot notation* is used so that defined module variables can be referenced by defining a label for the module instance and referencing the variable in the form *module.variable*. This feature is additive in that a module instance object can be defined inside modules such that a trail of dot notation can be used to define the context of the variable. This feature is extensively used in counterexamples to reference the actual values of variables declared in modules at a given state.

Example 4.2. *In gigamax.smv the module "memory" defines the variable "master" of type boolean. To reference master from main and find the value of master relative to m, we declare a process m of type memory and reference m.master.*

To trace the definition location of a variable in a NuSMV program, methods are required to analyse the modules variable and the string in the module variable dot notation representing a variable, and recursively search for each declared variable from the dot string until the module that originally defined it is found. Using the module and variable name the domain can be returned. This is necessary to derive domain information for counterexamples when building a local model. To handle this contingency, the *smvProgram* contains a member function *traceVariable* which can take a variable in dot notation and the module it is referenced in, (defaulting to the main module) and derive the original module the variable is defined in.

An example of this can be seen in the gigamax cache coherance program. In the main module there are five module declarations each with their own sub-variables declared within the modules. However in the counterexample, variables are referenced with their full module context, the full dot notation leading to the modules variable declaration.

This causes problems as it does not give any information about the domain of the variable in the counterexample. To solve this, a hash table mapping modules to the variables it defines and their respective domains is created such that variables can be traced back to the modules which created them to ascertain their type and domain.

Example 4.3. *In the NuSMV program gigamax.smv a variable "p0" of process type "processor" is defined in the main module. Processor inherits from the module "bus-device" which declares its own boolean variable "waiting", indicating the modules status. In the counterexample this variable would be represented with its valuation as p0.waiting = 0 without a reference to its status as a boolean variable. To determine the type of waiting, we pass the method traceVariable p0.waiting and main as the module name. Referencing the variable hash table for main, p0 is found to be a process type "processor" module. traceVariable recurses, applying the process to the processor module. No reference is found to waiting in the variable hash table but processor inherits from cache-device and bus-device. Applying the process to bus-device we find waiting is of type boolean and its domain can be returned to the calling function.*

4.2.3 Counterexample Parsing

When NuSMV returns from a model checking session it does so with the results of verification of the model and gives verbose output describing the violation and data pertinent to the checking session.

A returned counterexample will be a sequence of states listed with system variables assigned a valuation from its respective domain. Each state is assigned an identifier kept in the header (*e.g.* for a root state ->*State: 1.1*<-), with variable assignments trailing. In NuSMV each subsequent state after the initial, states are followed with variable valuations only where changes have occurred in the transition from the last state. This is a means of reducing the size of a counterexample, as it is possible to see smv files defining hundreds of variables, each needing to be defined with a valuation at a state.

For models defined with a process layer, inputs are defined with their own identifier and header. This describes the process selector variable which handles control of processes based on behavioural constraints (such as process fairness, defined in [15]). Counterexamples also report session information about the model checking procedure, such as which properties it satisfied, type of trace, trace description, system diameter and state reachability.

Further to this description, in the case that a counterexample trace is an infinite path, it will contain one or more indicators of where the loop begins (*i.e.* - - *loop starts here*). This indicates that a transition occurs from the final state in the counterexample to the state immediately following the loop delimiter, such that the loop is created. A characterisation of general counterexample form can be seen, such that we can parse generated counterexamples to determine if property specifications were found false. If so, build the counterexample into a tree-like structure which can be updated based on the unsatisfied properties.

With these characterisations of counterexamples, it is easy to see that there are challenges in building a parser for counterexamples. Firstly, labels from previous states in a sequence need to migrate over to following states if no change occurs. Further, we need to implement flags which indicate where SCCs exist and which state

a final state in an SCC transitions back to. Domain data for variables and granular details about the program being analysed are required to build the propositional atom set for the local model update process. Finally, session details and session meta-data better defining the violation can be collected and structured to give the developer more information about the session[4].

smvCounterexample Module

To address these points, the class *smvCounterexample* was designed with the purpose of the lexical analysis and parsing of counterexamples into a program manageable and human readable format. This class can be found in the *smvCounterexample* module, of the *pySMV* package included in the *l-Up* suite.

Algorithm 4.3: smvCounterexample class definition.

```
01: class smvCounterexample():
02:     def __init__ (self, filePath, programPath, debug = 0):
03:         self.filename = filePath.split("\\")[−1]
04:         self.modules = {}# hash table
05:         self.smvProgObj, self.stateCount = smvProgram(programPath), 0
06:     def analyseCounterexample(self, rawFiles)
07:     def initialiseStates(self, rawStates)
08:     def initialiseTransitions(self, trans)
09:     def translateCounterexampleToModel(self)
10:     def reduceFormula(self, form, templateList)
```

smvCounterexample takes as a first argument the file path to the derived counterexample, an smv program object to derive domain information from the model specification file, any debug flags and builds a local model based on a returned counterexample and the model specification file. As with the other created class objects in the *pySMV* package, *smvCounterexample* uses *smvTools* to provide regular expressions for extracting counterexample information. The regular expressions are applied based on smv syntax to extract the following metadata about a counterexample: the filename and extension, property valuation from the checking session, the trace description and type, program size and model state count. *smvCounterex-*

[4]Counterexamples from NuSMV sessions on the semaphore, gigamax cache coherance protocol and sliding window protocol are given in Appendix B to demonstrate parsing and update.

ample also translates the counterexample into a local model format which is more efficient and reduces the effects of time complexity. This will be discussed in the following sections.

Parsing Labels

In order to determine a valuation of a variable at each state of the counterexample, we need a method of carrying over a valuation from previous states, if a check is necessary we can reference the applicable valuation by referencing the state. *buildState()* has the duty of creating the label space for all states in the counterexample. This is done by iterating over labels in the parsed counterexample file and carrying over valuations on variables that have not changed. We can then assign states a new variable valuation if changes have occurred since the last transition.

Algorithm 4.4: *buildState*(self, stateSet, delta = 0).

```
01: def buildState(self, stateSet, delta = 0):
02:     varBases, keys, varBaseSave = {}, stateSet.keys(), {}
03:     keys.sort()
04:     self.initialState = keys[0]
05:     for state in keys:
06:         if delta: varBaseSave = {}
07:         for var in stateSet[state]:
08:             if delta:
09:                 self.varBuild(var, stateSet[state][var].strip())
10:             else:
11:                 varBaseSave = copy.deepcopy(self.varBuild(var, ...
12:                     ... stateSet[state][var].strip()))
13:             varBases[state] = varBaseSave
14:     return varBases
```

The method *varBuild()* is utilised to create a hash table which maps variables to their valuations for a given state, calling itself recursively if dot notation indicates the module context of the variable belongs to nested modules. When a complete hash table of the variables and valuations are created it is returned to *buildState()* and mapped to the state.

This is an efficient method for structuring valuations on variables such that the

correct module context is maintained for each variable, as many instances of the same module can be called, each with an instance of a variable with the same name. Dot notation maintains correct module context for these variables.

Algorithm 4.5: *varBuild*(self, variable, assign, varBase = {}).

```
01: def varBuild(self, variable, assign, varBase = {}):
02:     if variable.find('.') != -1:
03:         rside = variable[variable.find('.')+1:len(variable)]
04:         lside = variable[0:variable.find('.')]
05:         if lside not in varBase.keys():
06:             varBase[lside] = {}
07:         varBase[lside] = copy.deepcopy(self.varBuild(rside, ...
08:             ... assign, varBase[lside]))
09:     else:
10:         varBase[variable] = assign
11:     return varBase
```

Counterexample Regular Expressions

In *smvCounterexample.py* regular expressions are used in conjunction with the *isolateSections()* method to break the counterexample into delineated sections to be interpreted by seperate functions[5]. *isolateSections* takes as an argument a list of regular expression headers and footers, which if encountered in the file, builds into a list from the head and terminates the list if one of the footer regular expressions is satisfied for the current line.

4.2.4 Variable Data Types and Valuations

In the previous chapter many details were removed to simplify the update algorithm. This was to aid an understanding of the structure and semantics applied in mapping an update to a model based on the temporal formula. These included details regarding type handling, label representation and reduction of variables into propositional atoms bound to a domain and given a truth value.

[5]Regular expressions and *isolateSections()* are kept in the helper module *smvTool.py*.

Counterexample Header:	'*** [\W\w]'
Process Input Header:	'-> Input: ([\W\w]*) <-'
State Header:	'-> State: ([\W\w]*) <-'
Specification	'– specification ([\W\w]*) is ([\W\w]*)'
Trace Description Mode:	'Trace Description: ([\W\w]*)'
Trace Type Field:	'Trace Type: ([\W\w]*)'
Diameter Field:	'system diameter: ([\W\w]*)'
Reachable State Field	'reachable states: ([\W\w]*) out of ([\W\w]*)'
Variable Assignment Field:	'([\W\w]*) = ([\W\w]*)'
Loop Sentinel Line:	'– Loop starts here'

Figure 4.4: Regular expressions for counterexample extraction.

In NuSMV many types of variable can be declared, this is a method of syntax sugaring the model specification process to ease the process of applying and defining transitions based on variable valuations. This is in contrast to the theory of model update which allows valuations to be placed on atomic propositional atoms through a label function.

To solve this disconnection between theory and practice, we ground these compound structures into their propositional atom equivalents and maintain a template list of each grounded label which can be linked to a string of binary values mapped to a state. We can find the value of a variable at a state by finding the index of the mapped valuation and apply an interpretation based on the type of variable. In their simplest form variables consist of three pieces of information, the label representing the variable, the possible domain for the variable and the given current valuation for the variable at that state. We will list in the following section how each data type can be interpreted with each of these parts of variables in mind.

Boolean Type: The base case for variable representation. A boolean value can be represented as a propositional atom where its valuation is a single bit in the valuation string. The index for the label in the label template would match the index in the valuation string.

Interval Type: An interval is a defined type given as an integer value with a maximum and minimum bound[6], *e.g.* $months = 1 \ldots 12$; an interval can be given a

[6]As is well established in model checking literature combinations of domain size and variable cardinality are causal agents towards the model explosion problem [46]. For this

binary representation with a length based on its size (*i.e.* maximum minus minimum value). Each element of the domain can be represented by some valuation to each propositional atom.

Example 4.4. *Months could be represented with 12 elements; 0-11.* $binary(11) = length(1011) = 4$ *bits and the valuation* $months = 7$ *would be represented as a propositional formula* $months_8 \wedge \neg months_4 \wedge \neg months_2 \wedge \neg months_1$[7]*, or the valuation string 1000.*

Word Type: A word can be seen as an array of size equal to the number of bits it represents. Each bit can be referenced through an index. *e.g.* a word *flags* of size 5 could be represented with the valuation 10101 and $flags[2] == 1$.

Enumerable Type: Enumerable types can be treated in much of the same way as an interval integer type. Each member of the domain can be enumerated such that some valuation to a set of bits representing the domain derives a valuation; *e.g.* the enumerated variable $week = \{mon, tue, wed, thurs, fri, sat, sun\}$ can be represented with 3 bits (7 elements, 3 bits gives 2^3 possible bit combinations) and the valuation at the state $week = fri$ is 100, or $week_4 \wedge \neg week_2 \wedge \neg week_1$.

Array Type: Array types can be treated as a multiplied version of the other types. Each array when defined is given an interval value defining valid indices for the array and the associated type to the array, including it being able to be defined as an array of type array (for multi-dimensional array types).

The defined size of the array multiplies against the number of bits required to represent a variable of the given type at an index; *e.g.* The valuation of the array 0..2 of scores: 0..5 would be representable with $3 \times 6 = 18$ bits, and the valuations $scores[0] = 4, scores[1] = 2, scores[2] = 0$ is representable with the string 100010000, or the propositional formula $scores_{0,1} \wedge \neg scores_{0,2} \wedge \neg scores_{0,4} \wedge \neg scores_{1,1} \wedge scores_{1,2} \wedge \neg scores_{1,4} \wedge \neg scores_{2,1} \wedge \neg scores_{2,2} \wedge \neg scores_{2,4}$.

These conventions have the following effects on the update process. For the algorithm to determine which bit to apply the propositional atom to we need to ground

reason all integers defined in NuSMV require that an upper and lower bound be defined to increase model tractability and reduce the model size.

[7] As 0 is not an allowed valuation representation begins at 1, thus $months = 1$ is represented as $\neg months_8 \wedge \neg months_4 \wedge \neg months_2 \wedge \neg months_1$.

the compound propositions in the ACTL formula such that it can be interpreted by the local model update algorithm and effect an appropriate modification. We will give an example of this and the representation of the label set at an initial state in Example 4.5. Also a case exists where a proposition is required to not be true at a state. This equates to some other value from the domain be true for the variable in question and the original valuation be untrue. Following this persistence checking needs to occur to be certain another property was not violated.

Example 4.5. *Applying the reduction of variables to propositional atoms in this section, we use the gigamax.smv program in Appendix B, such that it can be interpreted by the l-Up framework. By analysing the main we can see that the program defines an enumerable type CMD with 9 elements passed to each module, and the 4 processes* p_0, p_1, p_2 *and* m. p_0, p_1 *and* p_2 *are defined as processors and inherit behaviour from bus-device and cache-device. These modules define the enumerable type variables state, snoop and cmd, and the boolean variables master, waiting and reply-stall.* m *is an instantiation of memory, which defines the boolean types master, busy and reply-stall, and the enumerable type cmd. Using the process detailed previously we reduce this variable to its corresponding set of atomic propositions.*

An enumerable type with 9 elements can be represented with 4 bits, boolean values with 1 bit and enumerable types with 3 elements can be represented with 2 bits. In total with each variable represented in each module each state can be represented using $4 + (3 \times (3 + 4 + 4)) + (3 + 4) = 44$ *bits*[8]. *Using the order given in the main, CMD would represent* $v[0]$-$v[3]$, *p0 represents* $v[4]$-$v[14]$, *p1* $v[15]$-$v[25]$, *p2* $v[26]$-$v[36]$, *and m* $v[37]$-$v[43]$.

The list of propositional atoms for this program are $AP = \{CMD_1,\ CMD_2,\ CMD_4,\ CMD_8,\ p_0.master,\ p_0.waiting,\ p_0.reply\text{-}stall,\ p_0.cmd_1,\ p_0.cmd_2,\ p_0.cmd_4,\ p_0.cmd_8,\ p_0.state_1,\ p_0.state_2,\ p_0.snoop_1,\ p_0.snoop_2,\ p_1.master,\ p_1.waiting,\ p_1.reply\text{-}stall,\ p_1.cmd_1,\ p_1.cmd_2,\ p_1.cmd_4,\ p_1.cmd_8,\ p_1.state_1,\ p_1.state_2,\ p_1.snoop_1,\ p_1.snoop_2,\ p_2.master,\ p_2.waiting,\ p_2.reply\text{-}stall,\ p_2.cmd_1,\ p_2.cmd_2,\ p_2.cmd_4,\ p_2.cmd_8,\ p_2.state_1,\ p_2.state_2,\ p_2.snoop_1,\ p_2.snoop_2,\ m.master,\ m.busy,\ m.reply\text{-}stall,\ m.cmd_1,\ m.cmd_2,\ m.cmd_4,\ m.cmd_8\}$.

[8]Not all of these states are reachable and using local models reduces the space of states further. Regularly, a 44-bit string could represent 1.7×10^{13} unique states.

4.2.5 ACTL Formula Reduction

As each variable declared in the NuSMV program and counterexample will be kept in a label template in a reduced propositional form, we need to correspondingly transform the propositional statements in the ACTL properties before the update search procedure to ease processing time. We replace each compound variable type with its propositional formula representation given above.

Example 4.6. *In the main module of the previously referenced gigamax program the property* $AG!(p_0.writable \ \& \ p_1.writable)$ *is expressed to stop two processors from writing to a cache simultaneously in any state of the program. With the list of propositional atoms defined in Example 4.5, the propositional formulas defined in the cache-device module for writable* ($writable := (state = shared) \ \& \ !waiting$) *and equivalences, we remove the program syntax sugaring and get the parsable formula* $AG((\neg(p_0.state = shared) \vee p_0.waiting) \vee (\neg(p_1.state = shared) \vee p_1.waiting))$. *We then reduce the formula further to valuation indices for each state to get the formula*

$$AG(\neg 11 \vee 12 \vee 5 \vee \neg 22 \vee 23 \vee 16),$$

where 11 *represents the atom* $p_0.state_1$, 12 $p_0.state_2$, 5 $p_0.waiting$, 22 $p_1.state_1$, 23 $p_1.state_2$, *and* 16 $p_1.waiting$.

Example 4.7. *To give an example of counterexample parsing we modify the gigamax.smv program to cause the system to violate the ACTL formula* $AG(\neg(p0.writable \wedge p1.writable))$. *To do this we have to have the two processors be in a state allowing both to write to a cache. On line 14 of the bus-device module of gigamax.smv we set waiting to FALSE if master is true and CMD is in a read-owned state. Passing this to NuSMV we receive a five state finite path counterexample explaining the violation, given in Figure 3 of Appendix B. Returning the counterexample location to an instantiation of smvCounterexample, initially it parses lines pertaining to the specification and the satisfaction of it in the model.*

If unsatisfied, it extracts the block of text beginning with the initial state and terminated by another counterexample declaration or the system diameter and reachable states footer. Regular expression grouping fields (i.e. ([\W\w])) assist in extracting the state labels and count (in use of initialiseStates() and initialiseTransitions()*

methods). Having extracted variables, labels are iterated over and types are found for each label from their respective module.

The resulting local model is described in the Kripke structure M below, where propositional atoms satisfied at the state are present and unsatisfied atoms are not present as a means of space saving. For the list of Atomic Propositions for this program we refer the reader to Example 4.5.

$S = \{s_1, s_2, s_3, s_4, s_5\}$, $T = \{(s_1, s_2), (s_2, s_3), (s_3, s_4), (s_4, s_5)\}$,

$L(s_1) = \{CMD_1, p_0.cmd_1, p_0.master\}$,
$L(s_2) = \{CMD_8, p_0.waiting, p_0.state_2, m.master, m.cmd_8, m.busy, p_1.cmd_1,$
$REPLY\text{-}WAITING, p_0.reply\text{-}waiting\}$,
$L(s_3) = \{CMD_1, p_1.master, p_1.cmd_1, p_0.writable, p_0.readable, p_0.state_1\}$,
$L(s_4) = \{CMD_8, p_1.waiting, p_1.state_2, m.master, m.busy, p_1.reply\text{-}waiting,$
$REPLY\text{-}WAITING, m.cmd_8, p_0.writable, p_0.readable, p0.state_1\}$,
$L(s_5) = \{CMD_1, p_2.master, p_2.cmd_2, p_1.writable, p_1.readable, p_1.waiting,$
$p_1.state_2, p_0.writable, p_0.readable, p0.state_1\}$.

4.3 pyFormula Package

The *pyFormula* package for *l-Up* was created for the purpose of having a stand alone package which could lexically analyse strings into structured tokens and parse formulae in universal computational tree logic. We can then reduce these formulae using known equivalence rules such that they can be used in the local model update algorithms. This has resulted in the creation of three modules central to *pyFormula*: *ACTLOps.py* for defining ACTL tokens and equivalence rules, *ACTLFormulaParser.py* for prototype ACTL formula parsing and *evalOperators.py* for definition of action tokens used later in constraint automata compliance. In all, this package contains 1078 lines.

To begin, we will look at ACTL token definition and briefly look at the ACTL parser. Further in this chapter we discuss action tokens.

4.3.1 ACTL Tokens

In the Python language all structures are treated as an object. For this reason we design ACTL formula tokens in an objective way such that classes of formula tokens are clearly defined for parsing. This includes making formula tokens uniquely identifiable, contain the correct amount of arguments and ideally, presentable in a fashion which represents correctly the formula in a string format. As a precondition to update semantics being applied to the model, the property formula must be passed as a well formed formula string by the user. An example of a not well formed formula is the property $\phi \equiv \mathrm{AF}(a\ \mathrm{U}\ b)$. This is not well formed because the universal quantifier is not defined for the innermost temporal operation. The correct way to define this is $\phi \equiv \mathrm{AF}(\mathrm{A}(a\ \mathrm{U}\ b))$.

In the *ACTLOps.py* module we define formula tokens as a class of operations *Op*, inheriting from Python lists and may take an undefined amount of arguments (n-ary).

```
class Op(List):
    def __init__(self, args):
        super(Op, self).__init__(args)
```

```
class AG(Op):
    def __str__(self):
        return 'AG(%s)' % tuple(self)
```

Figure 4.5: Type definitions for ACTL Formula Tokens.

In this definition Op inherits from the List type and begins by applying the initialisation of its super class, List. To define ACTL tokens from the Backus-Naur Form in chapter 2 we have each token derive from the super class *Op* and apply an ary count such that they accept a specific argument count and throw an exception if the wrong amount of arguments are passed. An example of this is the AG token. We define a new class AG which is unary. The *__str__* method is defined such that if the class object is treated as a string or is passed to standard output it will print its token type (*i.e.* "AG"), parentheses to establish its scope, then call the string method of its first argument, be it another formula token or an atomic proposition.

Another similar case is the binary operation disjunction, or $\vee$. In this case it again inherits from *Op* but its *__str__* method returns both of its arguments,

surrounded by parentheses and seperated by a dash, allowing a text representation for disjunction.

```
class Or(Op):
  def __str__(self):
    return '(% s) | (% s)' % tuple(self)
```

```
class AU(Op):
  def __str__(self):
    return 'AU(%s U %s)' % tuple(self)
```

Figure 4.6: Type definition for binary ACTL formula tokens.

This representation of ACTL formulas allows the nesting of ACTL formulas to create the required tree structure inherent in formulas where atomic propositions are leaves. Methods for modifying ACTL formulas or deriving information pertaining to a formula are kept in the *ACTLops.py* module in the $formulaeOperators$ package of the *l-Up* software suite. This includes means of performing transformations into disjunctive or conjunctive normal form, removal of double negatives, applying de Morgans laws, identification of specific formula tokens and proper formula formatting.

4.3.2 ACTLFormulaParser

Included in this package is the *ACTLFormulaParser.py* module which uses *lex* and *yacc* to take a string representing an ACTL formula, perform lexical analysis to identify ACTL tokens and parse the formula into a structure usable by the update algorithm. *ACTLFormulaParser* defines ACTL parsing rules in Backus-Naur Form, identifies parsing patterns and applies the appropriate rule to generate the nested token structure shown in the previous subsection. A formula string can be parsed by creating an instance of class *actlFormulaParser* and calling the method *stackACTLFormula()* with the formula string and any debugging flags.

It should be stressed that *ACTLFormulaParser.py* is a prototype and, while working for many classes of ACTL formulas it does not guarantee an automatic precedence of tokens. The author recommends using parentheses to force precedence in parsing to get the desired semantic effect.

4.4 pyModel Package

The central package to the *l-Up* environment is the *pyModel* package, containing modules which enact the algorithms described in Chapter 3 and contains approximately 3300 lines of code. Central to the *pyModel* Package is the *Model* module, containing the *Model* class which enacts update with $Update_c()$. The *Model* class inherits from the *Kripke* class defined earlier in the *Kripke* module. In this way *Model* can perform all of the model-like operations of *Kripke* but abstract away many of the details inherent in creating the Kripke structure such as transition structure and state labelling. The *Model* class can be seen as an abstraction layer on which update can occur.

Algorithm 4.6: Model constructor.

```
    class model(self):
01:     def __init__(self, model, init):
02:         self.states = model.states
03:         self.init = state("")
04:         self.transitions = model.transitions
05:         self.destTransitions = model.destTransitions
06:         self.SCCs, self.treeDepthDict = {}, {}
07:         self.rollBackStep, self.skipFlag= [], False
08:     def updateApply(self, updates)
09:     def update(self, s, formula, debug = 0)
10:     def updateProp(self, s, formula, debug = 0)
11:     def updateConjNested(self, s, formula, debug = 0)
12:     def updateDisjNested(self, s, formula, debug = 0)
13:     def updateLabelAX(self, s, formula, debug = 0)
14:     def updateLabelAF(self, s, formula, debug = 0)
15:     def updateLabelAU(self, s, formula, debug = 0)
```

4.4.1 Kripke Module

The Kripke class is defined to represent a Kripke structure and encapsulate the required methods which can be applied to the embedded transition structure, states and labels. An object of type Kripke allows manipulation of the structure, access to private attributes, and general Kripke structure internal attribute queries.

The Kripke Class allows two methods of accepting input. The simpler represen-

tation mimics the notation used for the simpler examples presented in this thesis, *i.e.* a set of states labelled with propositional atoms done so with a labelling function, and a set of tuples mapping the transition of one state to another. Beyond this, update can also be performed on SMV models reduced using the process discussed in the previous section of the chapter. Model checking can be applied to simple Kripke structures described in the examples provided earlier.

Example 4.8. *Entering a simple Kripke Structure* $M = (S, R, L)$*, where* $S = \{s_1, s_2, s_3, s_4\}$*,* $R = \{(s_1, s_2), (s_2, s_3), (s_3, s_3), (s_3, s_4), (s_4, s_4)\}$ *and* $L(s_1) = \{a,\ b,\ c\}$*,* $L(s_2) = \{h,\ g,\ i\}$*,* $L(s_3) = \{m\}$*,* $L(s_4) = \{a\}$*, for testing purposes in the Kripke class, can be represented as*

$$S = \{s1 : a; b; c,\ \ s2 : h; g; i,\ \ s3 : m,\ \ s4 : a\}$$
$$R = \{s1 : s2,\ \ s2 : s3,\ \ s3 : s3; s4,\ \ s4 : s4\}$$

Kripke Structure Representation

In Python many data structures come included as standard when the Python interpreter is installed. In this implementation we use the dictionary hash tables type to store model states as indices to the dictionary and have each index refer to a string of atomic label valuations associated to the given state. This has the benefit of $\mathcal{O}(1)$ access time for state label valuations and supports dynamic insertion and deletion of states and valuations. States are represented as their own class inheriting from the object class, each with an associated label and depth in the counterexample model. Labels in *l-Up* are represented as arrays of boolean types using the *NumPy* package of Python for implementing each states array of boolean valuations.

Example 4.9. *The initial state* s_0 *of some Kripke structure* M *has the label function* $L(s_0) = \{\neg i, j, \neg k\}$ *where* $AP = \{i, j, k\}$*. In the Kripke instantiation this is represented in the set of states self.states* $= \{s_0 : [False, True, False]\}$ *where the valuation of* j *can be accessed through self.states*$[s_0][1]$*, evaluating to True.*

In the implementation, relations between states are kept as two-way adjacency lists, implemented as hash tables mapping states to the states they transition to and a secondary destination table which maps the transitioned state back to its

origin (*i.e.* if $(s_i, s_j) \in R$, $(s_j, s_i) \in R_{dest}$). This allows backwards traversal through the tree-like model which is used for SCC detection and models which contain invalid transition structures. This method of representing relations was chosen over adjacency matrices as the corresponding transition structure for local models is a directed tree-like graph, and in general adjacency lists are more efficient for structures with lower relation saturation (where maximum saturation is S^2 for structures allowing loops).

With the structures defined, manipulations to the represented model can be done through a library of methods defined in the *Kripke* class. These methods enact modifications defined by the types of update tuples, including addition or removal of labels, states and relations. Further, the methods allow queries on the state of the model and structural details of the transition system and state. This also includes identification of SCCs and path traversal, which we will analyse in the following sections.

4.4.2 Strongly Connected Component Indexing

A special case for consideration in model structure is Strongly Connected Components (SCCs) defined in Chapter 2, Definition 2.4. SCCs constitute infinite paths in tree-like structures and require indication in the system algorithm such that when states are traversed the initial entry state, member states of the cycle, and paths leading from the SCC are identified. This is done primarily for satisfying temporal properties whose semantics require future satisfaction (*i.e.* AF, AU).

To address this challenge we apply a method of SCC indexing for tree-like models such that SSC states can be parsed, traversed and paths outwardly transitioning from the SCC can be identified. In this way, when a state in the model is found to be in the SCC path, for formulas whose semantics require satisfaction in infinite paths, operations can be applied for SCC cases and if necessary, its states in the SCC which have relations to other states not in the SCC paths can be operated upon.

For this purpose in implementation we design a class *SCC*, which is described by

the initial state of the SCC, the list of states in the SCC, the dictionary of relations describing the cyclic path, and a dictionary describing any paths coming from SCC states. *SCC* also defines methods allowing path search of states in the SCC and related functionality.

Algorithm 4.7: SCC constructor.

```
      class SCC():
01:       def __init__(self, entry, states = {}, sccTransitions = {},
              outTransitions = {}):
02:       self.states, self.sccTransitions = states, sccTransitions
03:       self.outTransitions, self.entryPoint = outTransitions, entry
04:       def retEntryArm(self)
05:       def findHandle(self, s)
06:       def inSCC(self, node)
07:       def traverseSCC(self, initial)
08:       def testSCCBranch(self, state, initNode)
09:       def retLastState(self)
10:       def retOutwardTrans(self)
```

Indexing SCCs

To ease computational costs associated with determining membership of a state to an SCC, SCCs are indexed prior to the update process. In this way states can be identified as being members of the set of states of the SCC in near constant time and each SCC can be treated as a path such that appropriate update semantics can be applied in regards to infinite paths.

Finding which states occur within an SCC is a case of searching the state space for state s, which has two relations in the set of transitions such for some s_x and s_y, $\{s, s_x, s_y | (s_x, s), (s_y, x) \in R \wedge s \neq s_x \wedge s \neq s_y\}$ (*i.e.* the state has two states transitioning to the state and neither are itself.). Having found such a state, we backwards traverse the path created by each of the states two incoming relations and determine which describes the cyclic path and which describes the path leading back to the root state (*lines 4 and 5*)[9]. Having determined which is the cyclic path

[9]as tree-like models are directed graphs, backwards searching through paths will inevitably lead to the initial state.

Algorithm 4.8: *indexSCCs().*

```
    def indexSCCs():
01:   for s in states:
02:     if hasOrigin(s):
03:       if len(destTransitions[s]) == 2 and not isSelfLoop(s):
04:         branchA = testSCCBranch(destTransitions[s][0], s)
05:         branchB = testSCCBranch(destTransitions[s][1], s)
06:         if branchA and branchB:
07:           return False
08:         elif branchA and not branchB:
09:           sccTrans, outwardTrans = ...
10:               ... retSCCStates(s, destTransitions[s][0])
11:         elif not branchA and branchB:
12:           sccTrans, outwardTrans = ...
13:               ... retSCCStates(s, destTransitions[s][1])
14:         else:
15:           return False
16:         SCCs[s] = scc(s, sccTrans.keys(), sccTrans, outwardTrans)
```

we iterate through the path, returning hash tables describing the cycle and paths descending from SCC states (*lines 9 and 11.*). Finally, each found SCC is mapped to its unique initial entry state into the hash table *SCCs* (*line 14*).

Algorithm 4.9: *testSCCBranch*(s, initState).

```
    def testSCCBranch(s, initState):
01:     while s != initState:
02:       if not hasSuccessor(s):
03:         return False #found initial state
04:       else:
05:         if hasOrigin(s):
06:           if predecessorCount(s) == 2 and s != initState:
07:             return False
08:           s = destTransitions[s][0]
09:         else:
10:           return False
11:     return True
```

testSCCBranch(): *testSCCBranch()* is a method called by *indexSCCs()* to determine if a path found describes a cyclic SCC path, a path descended from the root, or an invalid path. *testSCCBranch()* performs a linear search until it finds the initial state of the cycle (*line 1*), determines the current state is the initial (*line 2*) or it finds another SCC entry state, indicating it is an invalid path or a path leading to the initial state (*line 3*). If none of these conditions are met and the initial SCC state is found the path taken describes the cyclic path and the method returns true (*line 11*).

Algorithm 4.10: *retSCCStates*(entry, armNode).

```
      def retSCCStates(entry, armNode):
01:     sccTrans, outWardTrans = {}, {}
02:     next, sccTrans[armNode], last = entry, [entry], armNode
03:     while entry != armNode:
04:       if successorCount(armNode) > 1:
05:         for branch in transitions[armNode]:
06:           if branch != next:
07:             outWardTrans = ...
08:               ... appendToDict(outWardTrans, armNode, branch)
09:       next, armNode = armNode, destTransitions[armNode][0]
10:       sccTrans[armNode], last = [last], armNode
11:     return sccTrans, outWardTrans
```

retSCCStates(): Having found the path which describes the cyclic path of the SCC (*entry*, *armNode*), we iterate through states and build a hash table adjacency list describing the path and any other paths with originate from SCC states (*line 7*). This process then returns, giving hash tables describing the cycle and SCC paths (*line 10*).

Having indexed SCCs present in a tree-like local model we can determine cycle membership for a state in time $\mathcal{O}(1)$.

4.4.3 Path Traversal

For checking satisfaction of paths in universal computational tree logic, depth first traversal is ideal for explicit state models, as it allows individual states along a path to be iteratively searched and checked for satisfaction, such that we can determine where the violation of some specification occurs. Time complexity is well established for depth first traversal, in worst case $\mathcal{O}(|V| + |E|)$ with no state repetition.

Algorithm 4.11: *modelTraverse*(M, s, ϕ).

```
      class modelTraverse():
01:       def __init__(self, M, init):
02:           self.states = M.states
03:           self.transitions = M.transitions
04:           self.destTransitions = M.destTransitions
05:           self.SCCs = M.SCCs
06:           self.currState, self.toVisit = init, []
07:           self.rollBackStep, self.skipFlag= [], False
08:           self.treeDepthDict = M.treeDepthDict
09:       def next(self)
10:       def updateDepthBit(self)
11:       def skip(self)
12:       def retDepthBit(self)
13:       def retNonSelfTransition(self)
14:       def retNonArmOrigin(self)
15:       def retSCC(self)
16:       def retNonLoopStates(self)
17:       def retAllButGiven(self)
```

Knowing this, class objects can be created which take reference to a local model and the initial state and methods which return references to states transitioning from previous given states can be employed. We establish satisfaction of temporal formulas over paths in an iterative manner for both propositional and universal temporal formulas. Methods are included with the traversal object which allows iteration over states, iterative returning of states for processing, the ability to skip branches if satisfaction has been maintained for the path, and the ability to backtrack through states if necessary.

Path traversal in *l-Up* takes into account the structural considerations of tree-like models such as branching transitions from single states, state self loops and Strongly Connected Components in Kripke structures. Two methods which are key to the traversal process are *next()* and *skip()*.

Algorithm 4.12: *next().*

```
    def next(self) :
01:     s = self.currState
02:     if self.skipFlag:
03:         self.skipFlag = False
04:         return s
05:     self.updateDepthBit()
06:     if self.inSCC(s):
07:         scc = self.retSCC(s)
08:         if s in scc.outTransitions.keys():
09:             self.toVisit.append(scc.transitions[s][0])
10:             self.currState = scc.outTransitions[s][0]
11:         elif scc.sccTransitions[s][0] != scc.entryPoint:
12:             self.currState = scc.sccTransitions[s][0]
13:         elif self.toVisit:
14:             self.currState = self.toVisit.pop()
15:         else:
16:             raise StopIteration
17:     elif self.hasSuccessor(s):
18:         if self.isSelfLoop(s):
19:             self.toVisit.extend(self.retNonLoopStates(s))
20:             if self.toVisit:
21:                 self.currState = self.toVisit.pop()
22:             else:
23:                 raise StopIteration
24:         else:
25:             self.toVisit.extend(self.transitions[s])
26:             self.currState = self.toVisit.pop()
27:     elif self.toVisit:
28:         self.currState = self.toVisit.pop()
29:     else:
30:         raise StopIteration
31:     return s
```

next() - *next()* returns the reference to a state transitioning from the last state passed as an argument from the traversal object, or a previously unvisited state if every state in the previous branch was visited. This method always returns a unique state in the model, if the last state reference passed was a leaf, this method returns a reference to a state from an untraversed branch last encountered or throws a *StopIteration* exception, indicating the last state in the sub-tree has been traversed and the search space is exhausted (*line 30*). Similarly, if a SCC is traversed when the last state leading back to the first cycle state is reached, it is treated like the leaf case. Finally, *next()* also maintains the depth attribute of states as it traverses the model, by modifying the depth bit value relative to its immediate ancestor (*line 5*).

Algorithm 4.13: *skip().*

```
      def skip(self) :
01:       if self.toVisit:
02:           self.skipFlag = True
03:           self.currState = self.toVisit.pop()
04:       else:
05:           raise StopIteration
```

skip() - *skip()* raises a bit flag in the traversal object such that when the *next()* method is again called the following referred state is the beginning of an unchecked path. This method is used in the case where satisfaction of the current path has been established and further path traversal is unnecessary, thus *skipping* the current path and beginning another. Using the methods *next()* and *skip()* we can perform all required traversal over paths of a local model to ascertain the satisfaction of some formula token by a model path.

Depth Analysis

In update, state depth is used to determine if an applied set of minimal modifications is smaller than some other set of modifications is. In this way we need to maintain a map of depths of each state by initially traversing the local model and mapping the depth of each state relative to the root.

To initially generate the map of depths to states we use the structure traversal object defined in the previous section. Here we iterate through each state using *retDepthBit()* to find the depth. *retDepthBit()* takes into account factors involving

Algorithm 4.14: *buildTreeDepthDict*(self, init).

```
    def buildTreeDepthDict(self, init) :
01:     travObj = modelTraverse(self, init)
02:     try:
03:         while True:
04:             s = travObj.next()
05:             self.treeDepthDict[s.name] = self.retDepthBit(s)
06:             if s.name not in self.treeDepthDict.keys():
07:                 self.treeDepthDict[s.name] = s.depth
08:     except StopIteration:
```

tree-like structures and updates the hash table if a depth cannot be found. This includes checking for SCCs as depth does not apply to SCC states. This method also handles cases where the state is a leaf or has a self referencing relation. This process is handled in the Kripke structure module (*kripke.py*).

Algorithm 4.15: *retDepthBit(self, s).*

```
   def retDepthBit(self, s) :
01:  if s in self.treeDepthDict.keys():
02:     return self.treeDepthDict[s]
03:     elif self.inSCC(s):
04:         if self.retSCC(s).entryPoint == s:
05:             if self.predecessorCount(s) == 2:
06:                 return self.retDepthBit(self.retNonArmOrigin(s)) + 1
07:             return self.retDepthBit(self.retSCC(s).entryPoint)
08:     elif self.hasOrigin(s):
09:         if self.isSelfLoop(s):
10:             return self.retDepthBit(self.retNonSelfTransition(s)) + 1
11:         return self.retDepthBit(self.destTransitions[s][0]) + 1
12:     else:
13:  return 0
```

After update, state depth must be refreshed for states, as modification may change a states depth relative to the root. This can be done by rebuilding the depth dictionary using the previous function.

4.5 pyAutomata

In *l-Up*, we have designed modules for the handling of variable and action constraints in update. These modules are bundled in the *pyAutomata* package to be referenced for constraint use. Using these methods and structures we can gear our updates towards modifications which comply to constraints and satisfy given ACTL property formulas, as described in Section 3.3 of Chapter 3. For this task we created the *pyAutomata* package for constraint automata compliance.

In implementation, automata need to be manually created by the developer for the local model in this iteration of *l-Up*. Automata files can be specified in the following manner, described in Figure 4.7.

01: **VAR AUTOMATA** test1

02: **S**={s0: i=1; t=1, ⋯
s1: i=2; t=1, sv: *}

03: **F**={s0, s1}

04: **R**={s0 → s1:
⋯ Assign(‘i’, Add(‘t’, 1));
s0 -> s0: Query(Gt(‘i’,0));
s0 -> sv: Query(Lt(‘i’,0));
s1 -> s1: Query(Gt(‘i’, 0));
s1 -> sv: Query(Lt(‘i’,0));
sv -> sv: Star()}

05: **V**={t = 0..2: interval,
i = 0..2: interval}

01: **ACT AUTOMATA** test2

02: **S**={s1: Query(Equals(‘left’, 1)),
s2: Query(Equals(‘right’, 1)),
sv: Star()}

03: **F**={s1, s2}

04: **R**={s1 -> s2: ‘precedes’;
s1 -> sv: ‘next’;‘exclusive’;
s2 -> sv: ‘precedes’;
‘exclusive’;‘next’}

05: **V**={left: bool, right: bool}

Figure 4.7: Example variable and action constraint automata definitions.

This approach to constraint automata is based around the parsing of actions in these automata. In the following section we give an insight into how actions are identified and parsed.

4.5.1 Actions

In this implementation actions are defined as nested tokens representing behaviour between state labels. To determine if the labels mapped to some pair of states possibly corresponds to an action in an automaton, an action formula from an automaton is recursively broken down. Right variables (variables occurring to the right of an assignment) and propositions in the action are replaced with their value at the first state and the result of the action is compared against the variable values at the second state. In this way we can say the difference between the state labels, described by the state transition, corresponds to the described action in the automata. We say that the difference between labels at two states implies a specific action.

Our framework system allows the definition of the assignment token operator ($:=$), basic arithmetic tokens ($+$, $-$, $/$, $*$, $<$, $>$), logical tokens (*and, or, not, TRUE, FALSE*) and the query symbol (?). We refer the reader to Example 4.10 for a demonstration of action tokens in use and action compliance in state transition.

Example 4.10. *Suppose the transition between two states describes the assignment of some interval variable i the value from another interval value j plus 1. This could be represented as $Assign(i, Add(j, 1))$. Suppose there exist two states, s_k and s_l which describe valuations placed on the two variables i and j described previously. s_k describes the valuations of $i = 3$ and $j = 6$, whereas s_l describes the valuations $i = 7$ and $j = 6$. Replacing the variables which are being referenced for their value, we get $Assign(i, Add(6, 1))$, the formula is then recursively broken down and based on the semantics of each operation it is found that the second state holds the correct values.*

For a further example of allowable constraint actions see Case Study 3 in the following chapter. Next, we define the structures for representing constraint automata in the software package.

4.5.2 Variable and Action Constraint Automata

We define a class of variable automata designed to represent the theoretical construct described in Section 2.5 of Chapter 2 for constraint automata. As seen with local models, states are kept in a hash table with each state mapping to a set of labels corresponding to some state in a local model. Automata transitions are defined as adjacency lists with hash tables containing lists of values. Associated actions are represented as transition tuples mapping to a list of actions connecting the two states. An example would be the transition (s_1, s_2) mapping to the action $Assign(i, 2)$. We declare final states as a list and the violation, initial and current state as instances of states. Methods from *lines 6 - 10* are involved in generating the automata from file specification, and *lines 11 and 12* are used to determine when the current automata state can transition based on a passed action.

Algorithm 4.16: Variable automata object class.

```
01: class varAutomata:
02:     self.states, self.transitions = {}, {}
03:     self.destTransitions, self.actions, self.labels = {}, {}, {}
04:     self.savedBranch, self.branchCount, self.final = [], [], []
05:     self.violation, self.initial, self.current = "", "", ""
06:     def readModelFromFile(self, path)
07:     def interpretVarAutomataLine(self, line)
08:     def buildState(self, line)
09:     def buildTransition(self, line)
10:     def buildVariable(self, line)
11:     def transitionAuto(self, source, dest, model)
12:     def checkSat(self, model, state2, formula)
```

The object design for action constraint automata is similar, however states are labelled with actions occurring between states and transitions are labelled with input action constraint symbols. Actions occurring in an action automata state can be mapped to actions occurring in a local model between states. Methods are given in *actAutomata* which build the action constraint and determine compliance to actions in local models.

Algorithm 4.17: Action automata object class.

```
01: class actAutomata:
02:     self.actions, self.variableType, self.labels = {}, {}, {}
03:     self.variableDom, self.states self.transitions = {}, {}, {}
04:     self.destTransitions = {}
05:     self.violation, self.initial, self.current = "", "", ""
06:     def readModelFromFile(self, path)
07:     def buildState(self, line)
08:     def buildTransition(self, line)
09:     def buildVariable(self, line)
10:     def transitionAuto(self, source, dest, model)
11:     def checkSat(self, model, state2, formula)
12:     def nextSat(self, s, act1, act2)
13:     def futureSat(self, s, act, automataObj)
14:     def pastSat(self, s, act, automata)
```

4.5.3 Constraint Compliance

Based on the type of automata generated, different semantics are applied to determine if a local model complies to a constraint automata. Variable constraint automata require the compliance to rules describing values applied to variables in a model and the interaction between different variables. Action constraints enforce specific ordering on actions such that we can better control how interacting processes behave.

We noted earlier that the framework does not explicitly take into account actions between states; to approximate this we determine the difference between the labels of two states in a local model and check it against the necessary action formula. This is performed in the algorithm *checkSat*. We also need a method of checking the value of some variable in a constraint automata action as a base case for action compliance between states.

variableLookup: To be compatible with NuSMV types, many variable types can be represented in constraint automata. The base case for action compliance requires that we can find the value of a variable at a state from its state labels, such that we can compare valuations at states in a local model. *variableLookup* returns the

value of the variable at a state using its domain and type in the *pySMV* module *smvProgram* to find the value. This method can be used by method *checkSat* to apply action semantics using variable state valuations at a state.

Algorithm 4.18: checkSat(self, M, s_1, s_2, ϕ_a).

```
01: def checkSat(self, M, s1, s2, φa):
02:     if φa == v:
03:         return variableLookup(M, s1, φa)
04:     elif φa == Assign: #arg1 := arg2
05:         return variableLookup(M, s2, φa1) == φa2
06:     elif φa == True: return True
07:     elif φa == False: return False
08:     elif φa == Not: return not self.checkSat(M, s1, s2, φa2)
09:     elif φa == And:
10:         return self.checkSat(M, s1, s2, φa1)
                ··· and self.checkSat(M, s1, s2, φa2)
11:     elif φa == Or:
12:         return self.checkSat(M, s1, s2, φa1)
                ··· or self.checkSat(M, s1, s2, φa2)
13:     elif φa == Eq:
14:         return self.satEq(M, s1, s2, φa1, φa2)
15:     elif φa == lt:
16:         return self.checkSat(M, s1, s2, φa1)
                ··· < self.checkSat(M, s1, s2, φa2)
17:     elif φa == GtEq:
18:         return self.checkSat(M, s1, s2, Not(Lt(φa1, φa2)))
19:     elif φa == Add:
20:         return self.checkSat(M, s1, s2, φa1)
                ··· + self.checkSat(M, s1, s2, φa2)
21:     elif φa == Sub:
22:         return self.checkSat(M, s1, s2, φa1)
                ··· - self.checkSat(M, s1, s2, φa2)
```

checkSat: To determine if two states in a local model imply an action described in the variable or action automata, we parse an action at an automata state and check the label functions of two given states. If the difference in label functions between the two states corresponds to the parsed value, we return a signal indicating the difference implies the action. *checkSat* is a recursive method, it begins from the root of the action formula and recursively calls itself, resolving the semantics of each operation based on values passed for a variable at a given state in the local model. Actions handled in *checkSat* are discussed in Subsection 4.5.1.

Variable Constraint Compliance

For variable automata, compliance involves identifying states in the local model which correspond to states in the variable automata through labels, as described in Definition 2.9. The algorithm used in this implementation is *variableCompliance*(M, VC, s_m, s_i) presented in Algorithm 4.19.

Algorithm 4.19: variableCompliance(M, $\mathcal{VC}$, s_m, s_i).

```
01: def variableCompliance(M, VC, s_m, s_i):
02:     varStack = []
03:     try:
04:         travObj = modelTraversal(M, s_m)
05:         while True:
06:             s_new = travObj.next()
07:             if len(M, transitions[s_m]) > 1:
08:                 varStack.append(s_a)
09:             for s_anew in VC.transitions[s_a] :
10:                 if checkSat(M, s_m, s_new, VC.actions[s_a, s_anew]):
11:                     VC.transitionAuto(s_anew)
12:                     if s_anew == s_v:
                                return False, s_m, s_new, s_anew
13:             if M.isLeaf(s_new) and s_anew not in VC.final:
14:                 return False, s_m, s_new, s_anew
15:             elif M.isLeaf(s_new) and varStack:
16:                 s_a = varStack.pop()
17:             if M.isSelfLoop(s_new):
18:                 if not isPath(VC, s_anew, actList[s_new]):
19:                     return False, s_m, s_new, s_anew
20:             if M.isSCCstate(s_new):
21:                     if not isPath(VC, s_anew, ···
                        actList(SCC[s_new].transitions.keys())) ···
                        or vPath(VC, s_anew, actList):
22:                         return False, s_m, s_new, s_anew
23:             s_m = s_new
24:     except StopIteration:
25:         del travObj
26:     return True
```

Starting from the initial state of the local model and initial state of the variable automata, a depth-first traversal of the local model is performed. For each transition

in the local model, we use *checkSat* to determine which state in the variable automata to transition to, based on available relations (*line 08*). If the automata transitions to the violation trap state, we can say the local model does not comply to the constraint automata (*line 10*). If a leaf state is reached in the local model (*i.e.* no other existing relations), the current state of the variable automata has to be a final state. If not, the local model does not comply to the variable constraint automata. If a leaf state is final and visited in a local model, we check a previously unchecked local model branch and jump back to the state in the automata which corresponded to a previously unchecked state.

There also exists the case for cyclic behaviour, as tree-like local models allow infinite paths. If a relation exists between a state and itself, transitioning behaviour of the variable automata must be so that a final state must be visited once to comply to the variable automata. This is similarly the case with the infinite path made by an SCC. As the infinite path of the SCC is traversed, there must be a corresponding infinite path in the constraint automata such that an final state is infinitely traversed. If the SCC path is traversed and no final state in the variable constraint automata is visited, the local model does not comply to the variable automata.

An example of update using variable constraint automata will be given in Case Study 3 of the following chapter.

Action Constraint Compliance

As action constraint automata states are labelled with actions, constraint compliance is maintained by identifying actions in the local model and mapping them to action states in the automaton. Transitions between states in the automata are labelled with action input symbols which dictate interaction between actions. To ascertain that the local model action complies to the necessary constraints on how it interacts with other actions in the local model, the action automata transitions state based on ordering of actions in the local model along paths. If an action order is satisfied the action automata can transition. Compliance to the action automata is then dictated by the given automata state, rather final, non-final or violating. If the local model encounters a leaf state and the automaton is in a non-final state we say

the local model does not comply to the automaton. Similarly if transitioned to an automaton violation state the local model does not comply to the automaton. A side case for compliance is self loops and SCCs which constitute infinite paths in tree-like structures. If a self loop or SCC exists in the local model there must be some reachable state in the automata which is final.

As mentioned in Chapter 2, Section 2.5, three types of action input symbol exists for action constraints which occur between actions; *next*, *precedes*, and *exclusive*. To check compliance we need intermediary methods to ascertain satisfaction of actions. We use *checkNextSat*, *checkFutureSat*, and *checkPastSat*,

Algorithm 4.20: checkNextSat(self, M, s, ϕ_a).

```
01: def checkNextSat(self, M, s, φ_a):
02:     if M.isState(s) and M.hasSuccessor(s):
03:         for nextState in M.transitions[s]:
04:             if not self.checkSat(s, nextState, φ_a): return False
05:         return True
06:     return False
```

Next: *Next* is the base case for action constraint semantics. As described in Definition 2.9, if an action constraint automata state has a relation to another state, where the relation is labelled with a *next* input symbol, all immediate next actions which occur in a local model must perform the action at the referenced related state. This was described in terms of transition functions in Chapter 2.

Using the method *checkSat*, we can identify relations between states which satisfy the criteria for performing an action present in an action constraint. Having identified that a model state satisfies an action existing at a state in the action automata, if the constraint requires compliance with a next input symbol, all connected next actions are checked in the local model by using *checkSat*. If some immediatedly connected action does not satisfy next semantics, a violation flag is returned. Applying this, we can determine if some action occurs immediately after some action and transition the automata accordingly.

checkFutureSat: *checkFutureSat* requires that an action occur along all future paths. To ascertain that a model does this, a depth first search is performed from the states resulting from the action. Using *checkSat*, each branch is checked, if some

Algorithm 4.21: checkFutureSat(self, M, s, act).

```
01: def checkFutureSat(self, M, s, act):
02:     if s in M.states.keys():
03:       try:
04:           travObj = M.modelTraverse(s)
05:           while True:
06:               s = travObj.next()
07:               if isSelfLoop(s):
08:                   if not self.checkSat(s, s, act): return False
09:               if self.checkSat(M, s, s_new act): travObj.skip()
10:               else:
11:                   if not M.hasSuccessor(s): return False
12:                   elif M.hasOrigin(s):
13:                       if s in M.SCCs.keys():
14:                           state = M.checkCycleSat(s, act)
15:                           if not state: return False
16:                           for cycleState in ···
17:                               ··· M.SCCs.sccTransitions.keys():
18:                               if cycleState == state: break
19:                               if cycleState in M.outTransitions.keys():
20:                                   if not M.checkFutureSat···
21:                                       ···(M.outTransitions[cycleState], act):
22:                                       return False
23:         except StopIteration:
24:           del travObj
25:     return True
```

path is checked and no action exists which satisfies precedes semantics, a violation exists and False is returned to indicate a path existing without the action.

checkPastSat: One of the conditions of *exclusiveSat* is that some action is not before a given action. *checkPastSat* traverses back towards the root of the model to determine if the action occurred earlier in the path. This method takes into account the structural components of a tree-like model and completes a backwards traversal which can ascertain action compliance of an earlier path. As the structure is a tree-like model there will only be one path to check which possibly contains SCCs or self loops.

Algorithm 4.22: checkPastSat(self, M, s, act).

```
01: def checkPastSat(self, M, s, act):
02:     s_p = s
03:     s = M.destTransitions[s]
04:     while not self.checkSat(M, s_p, s, act):
05:         if M.isSelfLoop(s):
06:             if not self.checkSat(M, s, s, act): return False
07:             s_p = s
08:             s = M.retNonLoopState(s)
09:         else if M.inSCC(s):
10:             pathSCC = M.retSCC(s)
11:             if M.checkSCCSat(M, M.retSCC(s), s, act):
12:                 return True
13:             s_p = M.destTransition[pathSCC.entryPoint][0]
14:             s = pathSCC.entryPoint
15:         else:
16:             if s in M.destTransitions.keys():
17:                 s_p = s
18:                 s = M.destTransitions[s][0]
19:             else: return False
20:     return True
```

Using these action compliance checker functions, we can ascertain the ordering of actions occurring in a local model. We construct the method devised in Algorithm 4.23[10]. As with variable constraint automata, traversal involves beginning from the initial state of the local model and automata. As the local model is traversed in a depth-wise manner compliance to actions in connected states of the automata are checked. If a specific pair of states in a local model are found to comply to an action and its input symbol, the action automata transitions state. As with variable constraints if the local model reaches a leaf state and the automata is not at a final state, the local model violates the action automata. Similarly, if a violation trap state in the automata is reached the action order in the local model violates the action automata.

There also exists the case of self loops and SCCs in the local model transition

[10] *actionCompliance* makes reference to a generator *actionTraverser* on *line 3*. This is a wrapper to *modelTraverse* returning tuples of states representing an action. This is excluded based on its simplicity and similarity in structure to *modelTraverse*.

Algorithm 4.23: actionCompliance(M, $\mathcal{AC}$, s_m, s_i).

```
01: def actionCompliance(M, AC, s_m, s_a)
02:     try:
03:         actTrav = actionTraverser(M, s_m)
04:         while True:
05:             act = actTrav.next()
06:             possActStates = AC.transitions[s_a]
07:             for s_anew in possActStates:
08:                 for possSymbol in AC.actions[(s_a, s_anew)]:
09:                     if symbol == 'next':
10:                         if nextSat(M, act[1], s_anew):
11:                             s_a = s_anew
12:                     elif symbol == 'precedes':
13:                         if futureSat(M, act[1], s_anew):
14:                             s_a = s_anew
15:                     elif symbol == 'exclusive':
16:                         if futureSat(M, act[1], s_anew) and not
                                ··· pastSat(M, act[1], s_anew):
17:                             s_a = s_anew
18:             if isLeaf(act[1]) and s_a not in AC.final: return false
19:             if s_a in AC.violation: return False
20:     except StopIteration:
21:         del actTrav
22:     return True
```

structure. These are again handled like variable constraints. Self loops require that a final state is reachable in the automata by performing the action in the loop action. If no final state is reachable the local model does not comply to the automata. In a local model SCC there has to be a reachable final state in the automata accessible by performing the actions which exist in the path made by the SCC. If none exists the local model does not comply to the action automata. Included in the algorithm is the use of an action generator (*line 3*). This is a wrapper function over a model Traversal object that returns tuples of states corresponding to actions in a local model. The operations of the action generator has been omitted from the text based on its trivial nature.

Having methods for determining the compliance to action constraints in a local model allows us to guide update towards modifications which satisfy a given ACTL

property, while also complying to behaviour in a variable constraint. These methods can be used in parallel to model update techniques to the effect of better guiding update towards desired behaviour.

4.6 Summary

In this chapter we have looked at the central functionality of the packages contained in the *l-Up* local model update tool. This has given us the opportunity to provide insight into how the elements of the implementation work together to derive fixes to local models from models specified in the SMV language. These packages are open repositories of code which can be used for future development in the field of model checking and update.

Having given implementation details, we apply the update process to two case studies; the semaphore sharing scenario, demonstrating application to process starvation situations and the Sliding Window Protocol; showing application to a situation where a man-in-the-middle attack can cause interruptions in transmission. Finally, we extend the local model update technique with constraint automata to better constrain the approach and give an example in the SPIN environment involving mutual exclusion between two processes entering a critical region, and dictate update behaviour through variable constraint automata.

Chapter 5

Case Studies for Update

In this chapter we present three case studies that illustrate, with applications, the local model update theory and tools. To begin, in Section 5.1 we model an abstracted semaphore sharing scenario between two user processes and devise counterexample repairs, such that a local model satisfies fairness properties. Following this, in Section 5.2 we provide the case study of an abstracted representation of the Sliding Window Protocol (SWP), where we derive candidate repairs for a model, exhibiting a man-in-the-middle attack modifying regular module behaviour. Finally, we present a further case study showing the efficacy of local model update in Section 5.3, with the technique used in conjunction with constraint automata to demonstrate how inclusion of variable and action constraints reduces complexity and derives more applicable results. This is done in the SPIN model checking environment.

In the first two sections, formulas taking the form of $\mathrm{AG}(\neg p \vee \mathrm{AF}q)$ are used. This form of specification is useful, in that the truth value of some propositional atom in the model can be bound to the truth value of another propositional atom along all paths given some condition. These formulas are useful as a means of describing fairness conditions between processes sharing some resource (*In all cases, if some process goes into an entering state, it should eventually enter the critical region*) or describing communication in a channel (*If the sender has some specific value, then in all futures the receiver should at all futures hold this value*). This specification is also non-trivial to update some local model by. This is because each state available from the root state needs to satisfy the sub-formula $\neg p \vee \mathrm{AF}q$ and it is often the case that a smaller subset of modifications can be found to satisfy every unsatisfied state.

5.1 Case Study 1: Semaphore Sharing

In this subsection we will demonstrate the approach on a modified semaphore sharing scenario taken from the NuSMV examples repository [100], *semaphore.smv*. In Figure 5.1 we define the behaviour of a user accessing a semaphore to secure some resource.

```
01:  MODULE user(semaphore)

02:      VAR
03:          state : {idle, entering, critical, exiting};

04:      ASSIGN
05:          init(state) := idle;
06:          next(state) :=
07:              case
08:                  state = entering & !semaphore : critical;
09:                  state = critical : {critical, exiting};
10:                  state = exiting : idle;
11:                  1 : state;
12:              esac;
13:          next(semaphore) :=
14:              case
15:                  state = entering : 1;
16:                  state = exiting : 0;
17:                  1 : semaphore;
18:              esac;
```

Figure 5.1: Semaphore user definition module.

User takes a single boolean argument, *semaphore*, which indicates the occupied status of some resource. User also defines a variable *state*, indicating the status of the user in accessing the resource. This has the domain values *idle*, *entering* (*i.e.* attempting to access the semaphore), *critical* (*i.e.* holding the critical region), or *exiting*, (*i.e.* semaphore is released) and finally returning back to the *idle* state. In this program each users *state* is initialised as *idle*. Transitioning behaviour based on *semaphore* are also defined in this module, where the semaphore bit is set to *held* (*i.e.* 1) when the *user* is in the *entering* state and *released* when in the *exiting* state.

After defining the module which details how a *user* behaves when setting a

semaphore, we define the interaction between the *users* and the *semaphore* in the *main* (Figure 5.2) and assert properties we wish the model to satisfy. We define a *semaphore* (*line 3*) and initially set it to 0 (*released*). Also, we define two *user* processes *proc1* and *proc2* each taking the boolean *semaphore* as the argument.

01: **MODULE** *main*

02: **VAR**
03: semaphore : boolean;
04: proc1 : process user(semaphore);
05: proc2 : process user(semaphore);

06: **ASSIGN**
07: *init*(semaphore) := 0;

08: **SPEC** $AG(!(proc1.state = critical \ \& \ proc2.state = critical))$

09: **SPEC** $AG(proc1.state = entering \rightarrow AF(proc1.state = critical))$
10: $\& \ AG(proc2.state = entering \rightarrow AF(proc2.state = critical))$

11: **FAIRNESS** running

Figure 5.2: Semaphore main module.

Finally we declare two ACTL specifications we would like to hold true for the model (*lines 8 and 9*). Firstly we desire that at all states of the model there cannot be some state where both processes state have them be in the critical region (*i.e.* AG $!(proc1.state = critical \ \& \ proc2.state = critical)$). We also wish that when a user process goes into an *entering* state that eventually it will, under all cases, enter the *critical* region. This is represented using the ACTL formula AG$(proc1.state = entering \rightarrow$ AF$(proc1.state = critical))$ & AG $(proc2.state = entering \rightarrow$ AF$(proc2.state = critical))$.

Analysing the program as a whole, we can see *semaphore.smv* has two enumerable variables with a domain size of 4 (*idle, entering, critical, exiting*), and a single binary variable which has 2 states. This gives us a total state space of $4^2 \times 2 = 32$. With the model constructed and the desired properties specified we pass control to the NuSMV verification tool to determine satisfaction of the property and derive counterexamples to analyse.

5.1.1 Program Domain and Counterexample Extraction

Calling NuSMV, we derive a counterexample *semaphore.cex* after the model checker has found that the model did not hold the second property AG($proc1.state = entering \rightarrow$ AF($proc1.state = critical$)) & AG($proc2.state = entering \rightarrow$ AF($proc2.state = critical$)). The counterexample describes the path $1.01 \rightarrow 1.02 \rightarrow 1.03 \rightarrow 1.04 \rightarrow 1.05 \rightarrow 1.06 \rightarrow 1.07 \rightarrow 1.08 \rightarrow 1.09 \rightarrow 1.10 \rightarrow 1.05 \ldots$. containing the Strongly Connected Component (SCC) [24] $\ldots \rightarrow 1.05 \rightarrow 1.06 \rightarrow 1.07 \rightarrow 1.08 \rightarrow 1.09 \rightarrow 1.10 \rightarrow 1.05 \rightarrow \ldots$. The counterexample works as a witness to the model allowing *proc1* to hold an *entering* state and a path existing where *proc1* never goes into the *critical* state. In Figure 5.3, we give a transition graph showing the counterexample expressing the violation of the property[1]. Similarly, we note that NuSMV found that the model held the property AG(!($proc1.state = critical$ & $proc2.state = critical$)) and thus didn't require modification to satisfy it.

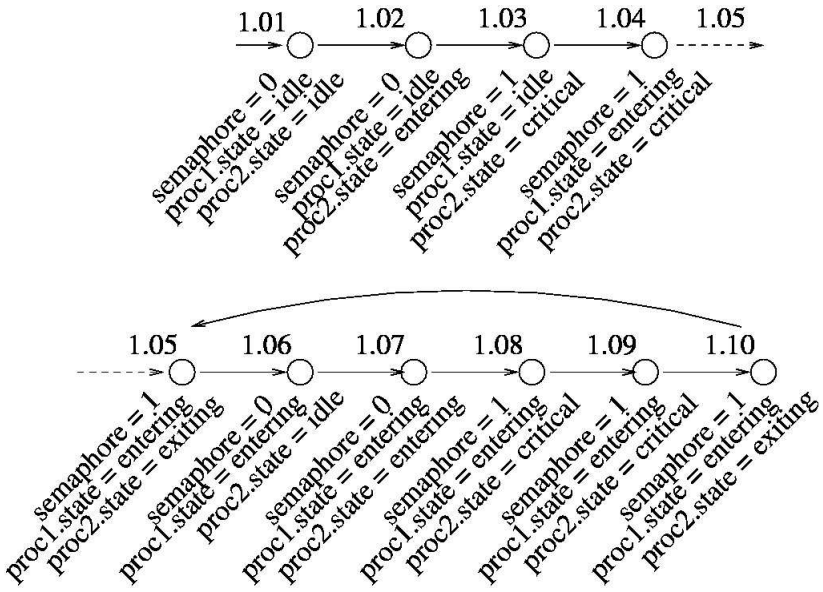


Figure 5.3: Counterexample for property.

With this, the counterexample can be represented by the Kripke structure $M = (S,\ R,\ L)$;

[1]See appendix C for the full counterexample provided by NuSMV for this update case study.

$$
\begin{aligned}
S = \ & \{1.01,\ 1.02,\ 1.03,\ 1.04,\ 1.05,\ 1.06,\ 1.07,\ 1.08,\ 1.09,\ 1.10\},\\
R = \ & \{(1.01, 1.02), (1.02, 1.03), (1.03, 1.04), (1.04, 1.05), (1.05, 1.06),\\
& (1.06, 1.07),\ (1.07, 1.08), (1.08, 1.09), (1.10, 1.05)\};
\end{aligned}
$$

$$
\begin{aligned}
L(1.01) = \{&semaphore = 0,\ proc1.state = idle,\ proc2.state = idle\},\\
L(1.02) = \{&semaphore = 0,\ proc1.state = idle,\ proc2.state = entering\},\\
L(1.03) = \{&semaphore = 1,\ proc1.state = idle,\ proc2.state = critical\},\\
L(1.04) = \{&semaphore = 1,\ proc1.state = entering,\ proc2.state = critical\},\\
L(1.05) = \{&semaphore = 1,\ proc1.state = entering,\ proc2.state = exiting\},\\
L(1.06) = \{&semaphore = 0,\ proc1.state = entering,\ proc2.state = idle\},\\
L(1.07) = \{&semaphore = 0,\ proc1.state = entering,\ proc2.state = entering\},\\
L(1.08) = \{&semaphore = 1,\ proc1.state = entering,\ proc2.state = critical\},\\
L(1.09) = \{&semaphore = 1,\ proc1.state = entering,\ proc2.state = critical\},\\
L(1.10) = \{&semaphore = 1,\ proc1.state = entering,\ proc2.state = exiting\},
\end{aligned}
$$

Looking at the counterexample, we can see that all states satisfy the specification that $proc2.state = entering \rightarrow \mathrm{AF}(proc2.state = critical)$ and from state 1.04 all states do not satisfy the formula $proc1.state = entering \rightarrow \mathrm{AF}(proc1.state = critical)$. This is because all states from 1.04 satisfy $proc1.state = entering$, but there is no state in the path which satisfies $proc1.state = critical$, as required by AF. To simplify the update process for the algorithm, we reduce the ACTL formula through equivalences and ground any variables bound to a domain into propositional atoms (an automatic process in the implementation prototype described in the previous chapter). Further, we replace propositional atoms in property formulas with the index of their valuation in propositional atoms as an encoding practice to aid space efficiency and speed in implementation.

Checking the domain for the variable *state* in the *user* module, we find it is an enumerable type with domain $\{idle, entering, critical, exiting\}$. As discussed in Subsection 4.2.4 of Chapter 4 these variables can be represented as propositional atoms in the update framework. As *state* has a domain size of 4, it can be represented with 2 propositional atoms[2]. In this way each valuation on the variables in the counterexample can be represented in terms of propositional atoms

[2]we subscript atoms with a numbering to uniquely identify them against other atoms representing the same variable.

$AP = \{proc1.state_1,\ proc1.state_2,\ proc2.state_1,\ proc2.state_2,\ semaphore\}$ using the original framework proposed for Kripke structures in Chapter 2. In this way, a valuation on the domain of proc1.state can be made by making valuations on $proc1.state_1$ and $proc1.state_2$. Applying a valuation of $proc1.state = critical$ would have an equivalent propositional atom representation of $\neg proc1.state_1 \wedge proc1.state_2$. This can be substituted into the property to complete the translation in the underlying system.

As an efficient method of representing valuations on the propositional atoms at each state, we reduce each variable to its propositional atom form and represent the valuation by applying a boolean valuation to each propositional atom. As the size and position of elements in AP will be the same for each state, we can represent these as contiguous blocks of true and false encodings for each variable. *proc1.state* can be represented with the binary mapping "$proc1.state = idle$" = 00, "$proc1.state = entering$" = 10, "$proc1.state = critical$" = 01, "$proc1.state = exiting$" = 11. The variable *semaphore* is defined as a binary variable and its domain can be represented using the mapping "$semaphore = 1$" = 1 and "$semaphore = 0$" = 0. We can construct a valuation string v for each state in the local model M, where $v[0] - v[1]$ represent *proc1.state*, and $v[2] - v[3]$ represent *proc2.state* and $v[4]$ represents *semaphore*. We then transform the label space to the labels:

$$L(1.01) = 00000,\ L(1.02) = 00010,\ L(1.03) = 00101,\ L(1.04) = 01101,$$
$$L(1.05) = 01111,\ L(1.06) = 01000,\ L(1.07) = 01010,\ L(1.08) = 01101,$$
$$L(1.09) = 01101,\ L(1.10) = 01111$$

In total, all valuations to labels for each state in the local model is represented in 50 bits.

In implementation we wish to have a fast method of accessing a valuation of some atomic proposition at a state. As we have the previous encoding for reducing variables to propositional atoms at a state, we can represent a value given to a variable by its propositional atom indices in the valuation string, given for each state in the model. Replacing variable valuations in ACTL properties with their propositional form, represented by their index in the valuation string makes access of a value for an atom at a state efficient and bridges the gap between the theoretical

system and types allowable in NuSMV. The atoms are represented in the formula by the index of the propositional atoms valuation at each state (*e.g.* if *semaphore = 0* was in the ACTL formula, $\neg(4)$ would replace it; if we checked state 1.09 by $\neg(4)$, $v[4]$ of 1.09 is *true*, thus $\neg(4)$ is not true.).

Applying this process to the ACTL formula, we would get the reduced formula $\mathrm{AG}((\neg 0 \wedge 1) \rightarrow \mathrm{AF}(0 \wedge \neg(1))) \wedge \mathrm{AG}((\neg 2 \wedge 3) \rightarrow \mathrm{AF}(2 \wedge \neg(3)))$. If we represented this in the reduced propositional atom form used for theory, we would get $\mathrm{AG}((\neg proc1.state_1 \wedge proc1.state_2) \rightarrow \mathrm{AF}(proc1.state_1 \wedge \neg(proc1.state_2))) \wedge$ $\mathrm{AG}((\neg proc2.state_1 \wedge proc2.state_2) \rightarrow \mathrm{AF}(proc2.state_1 \wedge \neg(proc2.state_2)))$ where 0 refers to the index of $proc1.state_1$, 1 to $proc1.state_2$, 2 to $proc2.state_1$, 3 to $proc2.state_2$, and 4 to *semaphore*. We apply equivalences to simplify the formula and bubble negation to propositional atoms ($\mathrm{AG}(\phi) \wedge \mathrm{AG}(\psi) \equiv \mathrm{AG}(\phi \wedge \psi)$, implication, de Morgans laws and double negation equivalences). This produces the formula $\mathrm{AG}(0 \vee \neg 1 \vee \mathrm{AF}(0 \wedge \neg(1)) \wedge 2 \vee \neg 3 \vee \mathrm{AF}(2 \wedge \neg(3)))$. Based on this, we can apply the reduced formula to M via $Update_c()$ and derive a minimal modification.

5.1.2 Deriving Update

We pass the local model, current state and unsatisfied formula to the function $Update_c()$. $Update_c()$ recognises the root formula being AG and passes control to the function $Update_{AG}()$ to apply its semantics to the underlying model.

Next, $Update_{AG}()$ applies the sub-formula to each connected state using $Update_c()$ from the current and saves the results. Semantics for $Update_\wedge()$ are used and for the second subformula of $\wedge$ we find that for every state in M, $(M, s) \models 2 \vee \neg 3 \vee \mathrm{AF}(2 \wedge \neg(3))$, and no modification is required based on this subformula (2 and 3 represent the indexed valuation string locations for the values of *proc2.state which satisfy the property at every state*). For the first subformula $0 \vee \neg 1 \vee \mathrm{AF}(0 \wedge \neg(1))$ $Update_\vee()$, is called.

The states *1.01*, *1.02*, and *1.03* each satisfies either 0 or $\neg 1$ (representing $\neg proc1.state = entering$). Beyond these, every state satisfies *proc1.state = entering* but has no connecting path satisfying *proc1.state = critical*. For these seven states

the possible updates include substituting the current state, such that the valuation at index 0 is true or 1 is false, or substituting a future state 1.06 such that it satisfies both 0 and $\neg 1$.

In $Update_{AG}()$, these updates are compared based on weak bisimulation semantics and it can be seen that, in choosing one update to satisfy each state such that we get the most minimal modification to a model, $Update_{AG}()$ selects $\mathcal{U} = \{(1.06, +, 0) \wedge (1.06, -, 1)\}$, as this will satisfy the formula at each unsatisfying state. This is done substituting a state with a new state satisfying the valuation on the two atoms. This is also optimal by weak bisimulation semantics as SCCs states have no notion of depth and if any branches exist in the SCC the formula will have already been satisfied in the SCC, satisfying any other path.

$$
\begin{array}{ll}
\textbf{1.04}: & \mathcal{U} = \{(1.04, +, 0) \vee (1.04, -, 1) \vee \{(1.06, +, 0) \wedge (1.06, -, 1)\}\} \\
\textbf{1.05}: & \mathcal{U} = \{(1.05, +, 0) \vee (1.05, -, 1) \vee \{(1.06, +, 0) \wedge (1.06, -, 1)\}\} \\
\textbf{1.06}: & \mathcal{U} = \{(1.06, +, 0) \vee (1.06, -, 1) \vee \{(1.06, +, 0) \wedge (1.06, -, 1)\}\} \\
\textbf{1.07}: & \mathcal{U} = \{(1.07, +, 0) \vee (1.07, -, 1) \vee \{(1.06, +, 0) \wedge (1.06, -, 1)\}\} \\
\textbf{1.08}: & \mathcal{U} = \{(1.08, +, 0) \vee (1.08, -, 1) \vee \{(1.06, +, 0) \wedge (1.06, -, 1)\}\} \\
\textbf{1.09}: & \mathcal{U} = \{(1.09, +, 0) \vee (1.09, -, 1) \vee \{(1.06, +, 0) \wedge (1.06, -, 1)\}\} \\
\textbf{1.10}: & \mathcal{U} = \{(1.10, +, 0) \vee (1.10, -, 1) \vee \{(1.06, +, 0) \wedge (1.06, -, 1)\}\}
\end{array}
$$

Figure 5.4: Possible types of update on *semaphore* to satisfy $AG\phi$.

Applying these modifications to the underlying local model with $Update_{apply}()$ we derive the local model fix

$$
\begin{aligned}
S = &\{1.01,\ 1.02,\ 1.03,\ 1.04,\ 1.05,\ 1.06,\ 1.07,\ 1.08,\ 1.09,\ 1.10\}; \\
R = &\{(1.01, 1.02),\ (1.02, 1.03),\ (1.03, 1.04),\ (1.04, 1.05),\ (1.05, 1.06), \\
&\quad (1.06, 1.07),\ (1.07, 1.08),\ (1.08, 1.09),\ (1.10, 1.05)\};
\end{aligned}
$$

$$
\begin{aligned}
&L(1.01) = 00000,\ L(1.02) = 00100,\ L(1.03) = 00011,\ L(1.04) = 10011, \\
&L(1.05) = 10111,\ L(1.06) = 01000,\ L(1.07) = 10100,\ L(1.08) = 10011, \\
&L(1.09) = 10011,\ L(1.10) = 10111.
\end{aligned}
$$

Translating the modification to $L(1.06)$ back from the binary notation to variable form, we have the modification $L(1.06) = \{proc1.state = critical, proc2.state = idle, semaphore = 0\}$, and we can see the applied modification in Figure 5.5. We

can also see that this new local model fix satisfies the primary property originally posed (AG !($proc1.state$ = $critical$ & $proc2.state$ = $critical$)). If necessary, this property can also be given to the $Update_c()$ algorithm with the model fix to make certain the new updated local region satisfies both properties.

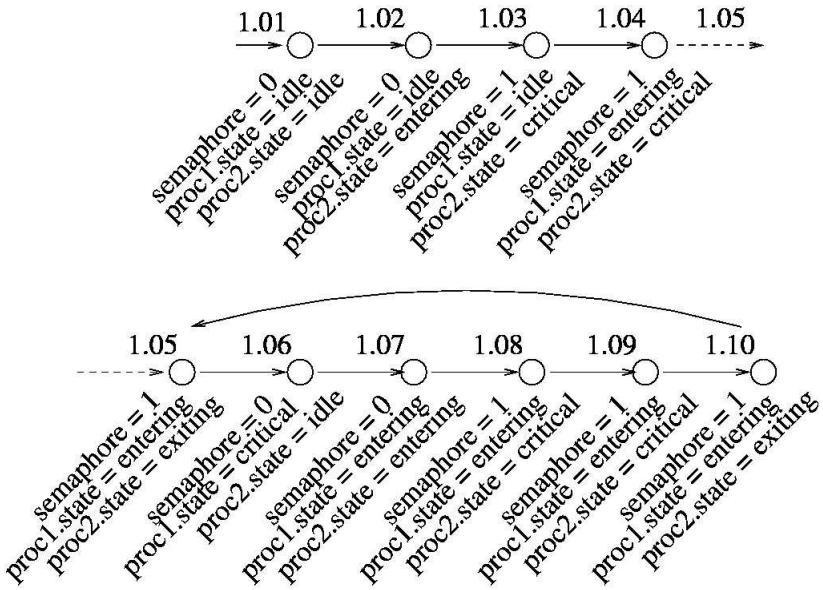


Figure 5.5: Local model fix for counterexample.

5.1.3 Results

Looking at the consequence of update, we can see faults inherent in the approach. Analysing the transition block for a user process in the program, many conditions must be satisfied before entering the critical section. Primarily the semaphore flag must be set; when leaving the critical section a graceful exit is expected by the process (looking at the updated local model, we see that *proc1.state = exiting* is not true at state 1.07). To better tailor our updates we need to take into account domain information in the process. In Section 5.3, we show how declared action and variable constraints can be a means of better guiding the update process in conjunction with the given temporal properties. In the following section we give another example of update and give a better analysis of worse case time complexity.

5.2 Case Study 2: Sliding Window Protocol

To demonstrate an application of local model update, we apply the process to an abstracted model of the Sliding Window Protocol (SWP) containing an attacker modifying data packets sent to a receiver. We derive a minimal fix to the counterexamples derived from the checking of the transmission protocol model[3].

In SWP the process occurs between three entities: a sender, a receiver and their given medium. Messages with unique identifiers are passed through the medium to the recipient, who then sends an acknowledgement token back through the medium along with the original messages identifier. Both sender and receiver maintain state by keeping a copy of the last identifier value sent. In the following section we will give a deeper insight into SWP.

5.2.1 Background

Sliding window is a common protocol used for point to point packet-based communication in networked environments. The protocol is used to maintain limited state between a sender and receiver, such that messages can be sent by one member and the other returns an acknowledgement of the messages receipt. Notable examples of Sliding Window Protocol variants are Point-to-Point Protocol (PPP) and general TCP connections [49].

The window between the sender and receiver is the space left between the amount of messages sent and the individual acknowledgements per message received back. Once an acknowledgement is received back from the message receiver, the sender's window slides forward a unit. Similarly, when a message is received from the sender the receiver's window slides forward a unit. When the maximum window size per identifier is reached the identifier rolls back to the beginning, thus the window of messages and acknowledgements between sender and receiver is a type of circular queue. In standard SWP the window size is determined by the amount of packets lost in transmission. The receipt of packets by the receiver dictates window size. If

[3]This case study was originally published in [102].

a packet is not received by its end-point within an allocated time it is considered a timed out and in the next sliding window round window size is reduced. On the other hand, window size is increased if packets are sent and acknowledged without timing out.

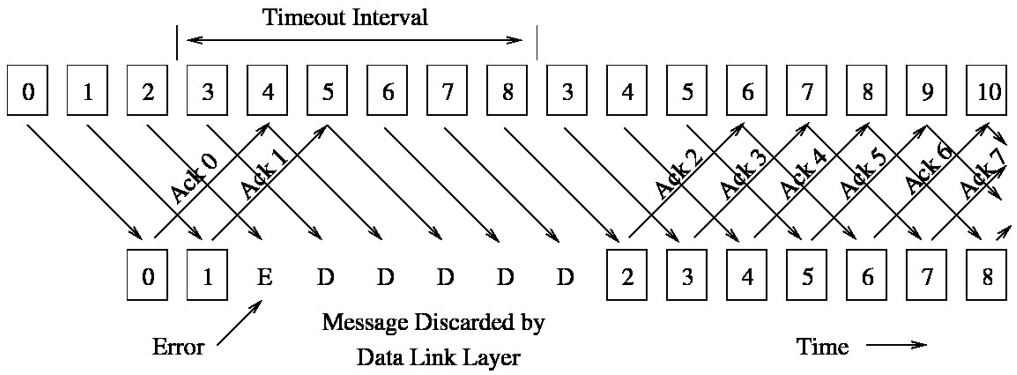


Figure 5.6: *Go-back-n* Sliding Window Protocol.

As packets can be lost in the medium a method for handling data loss is required. Sliding window has two methods for handling errant packets, *go-back-n* and *selective reject*. In *Go-back-n*, as shown in Figure 5.6, the receiver window size is kept at 1 such that if a message packet is lost the receiver stops sending acknowledgements until a message with the correct identifier is received. The sender will eventually retransmit from the original lost frame and when accepted by the receiver, the receiver will return an acknowledgement. The other method, *selective reject* has the receiver resend the last acknowledgement so the sender knows to resend the packet, which occurs after the acknowledgement with the old identifier. In sliding window protocol variables such as link speed, window and packet size affect elements such as medium congestion and saturation.

5.2.2 Methodology

The applied method for determining the worst case complexity and efficacy of local update is similar in scope to the previous case study, however a more complex variant on the property is used and the case study is conducted with running time and resource use in mind, over process explanation. This case study involves modelling SWP by iteratively increasing window size to determine computation time for update

as model and formula size increases. For the purposes of this test we check window size ranging for $n = 2 \ldots 8$ and log computation time. By increasing window size we increase the variable domain for the variables, representing sent and received packets and by connection the total state count. Window size has an n^3 relationship to model state count (*i.e.* a window sizes of range $2 \ldots 8$ yield a model state count ranging from 8 to 512, based on declaring three interval values). This is related to the propositional atoms generated through the interval values, however in practice many of these states are not reachable.

Another point to note about the update process, due to local models being, by definition, a subset of states from the original which, starting from the root state, lead up to the state violating the property, it is the case that every state within the tree-like model will be a reachable state. In the previous iteration of model update many of the states of the resultant model would not be reachable from the initial state.

For the purpose of modelling, we use the NuSMV language. We define a liveness property we wish to check based on the window size specified. We apply equivalences to specifications given and perform preprocessing to the model and for each successive model we find $(M, s) \not\models \phi$ and derive a counterexample C with which we apply $update_c(C,\ \phi)$, such that $C \models \phi$. All experiments are completed on a system with an AMD Phenom 8450, 2.10GHz, 3-core processor and 4GB 800MHz RAM.

5.2.3 Modelling Sliding Window

For the purpose of simplicity we abstract out many of the details of the protocol to verify a single property in the model. We have also assumed a perfect transmission medium and have abstracted issues of the transmission medium from the domain. For the purposes of the session we ignore the packet being sent and only model the message identifier. Finally, in true sliding window protocol window size is determined by the count of data failures in sending a packet from the sender to receiver. If some x failures occur, the window size is reduced. For the purpose of this abstract model this feature has also been removed, and the window size is varied to study complexity.

We define two processes, the *sender* and *receiver* which accept two arguments representing the packet identifier *sent* and *received* either way, each called *sendId* and *receivedId* respectively, and the window size for the current modelling session[4]. The incrementing of each identifier represents the shifting window buffer performed on the sender module when it receives an identifier value one higher than its last sent message. With this it sends out a message with the next identifier.

```
01: MODULE sender(receivedId, n)
02:   VAR
03:     sentId : 0...(n − 1);
04:   ASSIGN
05:     next(sentId) :=
06:       case
07:         (sentId + 1) mod n = ...
08:         ...receivedId :
09:           ((sentId + 2) mod n)...
10:           ...union sentId;
11:         1 : (sentId + 1) mod n;
12:       esac;
13:   FAIRNESS running
```

```
01: MODULE receiver(sentId, n)
02:   VAR
03:     receivedId : 0...(n − 1);
04:   ASSIGN
05:     next(receivedId) :=
06:       case
07:         (sentId = receivedId) : ...
08:         ...receivedId;
09:         1 : ((receivedId + 1) mod n);
10:       esac;
11:   FAIRNESS running
```

Figure 5.7: Sender and Receiver module.

The receiver sends an acknowledgement when it receives an identifier higher than the last message, returning the identifier received. As in traditional SWP, when the window buffer is full and all acknowledgements have been received a new window is established, rolling back the identifier to zero and sending the next set of data. In the SMV model, this is represented with the modulo (*line 8* of the *receiver* module).

To cause the model to violate the given property we introduce a *man-in-the-middle attacker* which takes the identifier and returns it altered, as shown in Figure 5.8. Here, we can see that the *attacker* takes two arguments (*line 2*), the first being the *interceptId*, and at *line 8* modifies the send identifier so that the receiver will never receive an identifier of 2, forcing a time out. Finally, we declare how these processes interact in the variable declaration of the main module and establish an initial value for the variables of the process, setting each value to start with the identifier 0 (*lines 9-11*).

[4]For the sake of simplicity we use a window size $n = 5$.

```
01: MODULE attacker ...                01: MODULE main
02:       ... (interceptId, n)         02:
03:   VAR                             03:   VAR
04:     returnId : 0..n;               04:     s : process sender(r.receivedId, n);
05:   ASSIGN                           05:     a : process attacker(s.sentId, n);
06:     next(returnId) :=              06:     r : process receiver(a.returnId, n);
07:       case                         07:
08:         (interceptId = 2) : 4;     08:   ASSIGN
09:         1 : interceptId;           09:     init(s.sentId) := 0;
10:       esac;                        10:     init(r.receivedId) := 0;
11:   FAIRNESS running                 11:     init(a.returnId) := 0;
```

Figure 5.8: Attacker and Main modules.

With a completed model, we move onto establishing desired properties which should be inherent in model execution and verification. Following this, we apply NuSMV and derive counterexamples explaining the violations of the property.

5.2.4 Properties and Checking

In this representation of the protocol we wish to establish a liveness property over the identifier being sent through the transmission medium. We say that at every state in the model on every computational path and for every value of the window n sent (*s.sentId*) implies that it will always be eventually received (*r.receivedId*). We express this using the ACTL specification:

$$\bigwedge_i \mathrm{AG}(s.sentId_i \rightarrow \mathrm{AF}(r.receivedId_i)).$$

With the model and property clearly defined, we apply a model checking session in *NuSMV* to ascertain the satisfaction of the property in the model. The model description for each differing window size is defined in its own file with the SMV file extension. The file names are appended with the window size attributed to the given file such that each SMV model file can be identified by its window size. For each SMV model file the maximum window size is defined and the property is expanded such that the property can be interpreted for the checking session. An example of this for window size five in NuSMV:

$$\begin{aligned}\text{SPEC AG}(&(!(s.sentId = 0)|(\text{AF}(r.receivedId = 0)))\&\\ &(!(s.sentId = 1)|(\text{AF}(r.receivedId = 1)))\&\\ &(!(s.sentId = 2)|(\text{AF}(r.receivedId = 2)))\&\\ &(!(s.sentId = 3)|(\text{AF}(r.receivedId = 3)))\&\\ &(!(s.sentId = 4)|(\text{AF}(r.receivedId = 4))))\end{aligned}$$

For each iterative window size *NuSMV* is called using the system command

NuSMV -r slidingwindow$n$.smv > swpcex$n$.cex

The result from checking the property is returned to *swpcexn.cex*, where n is the window size. The session result file is then parsed and the result is checked. The counterexample returned from NuSMV is extracted and built into a Kripke structure which can be traversed and have $Update_c$ be applied to effect modification.

To simplify the property for program use we use equivalences and transform it to DNF. This allows the update algorithm to evaluate negation as applied to propositional atoms and evaluate possible update sets as disjunctions of terms, turning the update process into a search problem for the most minimal set of updates, as dictated by the weak bisimulation guided minimal change semantics. In the case of a window size equal to five, this explodes the count of propositional atoms in the formula from 10 to 160 and creates 128 terms connected by 31 nested clauses.

$$\begin{aligned}&\text{AG}((\neg(s.sendId = 0) \wedge \neg(s.sendId = 1) \wedge \neg(s.sendId = 2)\\ &\wedge \neg(s.sendId = 3) \wedge \neg(s.sendId = 4)) \vee \ldots \vee (\text{AF}(r.receivedId = 0)\wedge\\ &\text{AF}(r.receivedId = 1)\wedge \text{AF}(r.receivedId = 2) \wedge \text{AF}(r.receivedId = 3)\wedge\\ &\text{AF}(r.receivedId = 4)))\end{aligned}$$

Figure 5.9: Formula transformed using DNF with $n = 5$.

5.2.5 Counterexample to Kripke Structure Translation

Having called *NuSMV*, we receive a report of the model checking session on the model[5]. For window size $n = 5$ NuSMV returns the counterexample representing the path $1.01 \rightarrow 1.02 \rightarrow 1.03 \rightarrow 1.04 \rightarrow 1.05 \rightarrow 1.06 \rightarrow 1.07 \rightarrow 1.08 \rightarrow 1.09 \rightarrow$

[5]See Section B.3 of Appendix B for the full counterexample for window size $n = 5$ referenced in this case study, or the examples section of *l-Up* for other values of n.

$1.10 \rightarrow 1.11 \rightarrow 1.08\ldots$ which contains the SCC $\ldots \rightarrow 1.08 \rightarrow 1.09 \rightarrow 1.10 \rightarrow 1.11 \rightarrow 1.08\ldots$. This counterexample is a witness to the fact that for multiple values of $s.sentId$ becoming true there doesn't exist a corresponding value for $r.receivedId$ in any future which equals the value of $s.sentId$. This counterexample trace is translatable to the tree-like Kripke structure $C = (S, R, L)$

$$
\begin{aligned}
S = \{&1.01, 1.02, 1.03, 1.04,\\
&1.05, 1.06, 1.07, 1.08,\\
&1.09, 1.10, 1.11\};\\
\\
R = \{&(1.01, 1.02), (1.02, 1.03),\\
&(1.03, 1.04), (1.04, 1.05),\\
&(1.05, 1.06), (1.06, 1.07),\\
&(1.07, 1.08), (1.08, 1.09),\\
&(1.09, 1.10), (1.10, 1.11),\\
&(1.11, 1.08)\};
\end{aligned}
\qquad
\begin{aligned}
L(1.01) &= \{s.sentId = 0,\ r.receivedId = 0\},\\
L(1.02) &= \{s.sentId = 1,\ r.receivedId = 0\},\\
L(1.03) &= \{s.sentId = 2,\ r.receivedId = 0\},\\
L(1.04) &= \{s.sentId = 2,\ r.receivedId = 1\},\\
L(1.05) &= \{s.sentId = 2,\ r.receivedId = 2\},\\
L(1.06) &= \{s.sentId = 2,\ r.receivedId = 3\},\\
L(1.07) &= \{s.sentId = 2,\ r.receivedId = 4\},\\
L(1.08) &= \{s.sentId = 3,\ r.receivedId = 4\},\\
L(1.09) &= \{s.sentId = 0,\ r.receivedId = 4\},\\
L(1.10) &= \{s.sentId = 1,\ r.receivedId = 4\},\\
L(1.11) &= \{s.sentId = 2,\ r.receivedId = 4\}.
\end{aligned}
$$

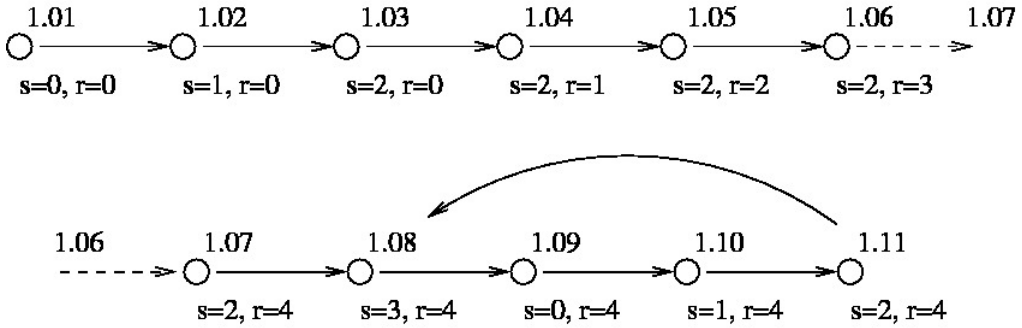


Figure 5.10: Local model for window size $n = 5$. s represents $s.sentId$ and r $r.receivedId$.

Using the script class *smvCounterexample* and *smvProgram* from the previous chapter, we find that *s.sentId* and *r.receivedId* are interval values. For the current program the interval size for variable domains is 0..4 (*0, 1, 2, 3, 4*). As in the previous case study, we can reduce these variable labels into a propositional atom format to perform operations upon and correspondingly, the property. For these labels, we find that each interval value can be represented with 3 bits each (5 elements in the domain, 2^2 can represent 4 elements, 2^3, 8 elements).

Convention 5.1. *Here, we enlist a notation for labels which only shows propositional variables in the label space which are true, for the sake of brevity. The set of atomic propositions for the model domain with a window size* $n = 5$ *is*

$$AP = \{s.sentId_1,\ s.sentId_2,\ s.sentId_4,$$
$$r.receivedId_1,\ r.receivedId_2,\ r.receivedId_4\}.$$

Here, the label function $L(1.08) = \{s.sentId = 3,\ r.receivedId = 4\}$ *is represented in the underlying system as* $L(1.08) = \{s.sendId_1, s.sendId_2, \neg s.sendId_4, \neg r.receivedId_1, \neg r.receivedId_2, r.receivedId_4\}$. *In other words, the valuation 3 on* $s.sentId$ *can be asserted with the propositional formula* $s.sendId_1 \wedge s.sendId_2 \wedge \neg s.sendId_4$. *The property used to update the model at the state becomes* $\neg s.sentId_1 \vee \neg s.sentId_2 \vee AF(r.receivedId_1 \vee r.receivedId_2)$. *To represent these valuations in implementation, we can reduce the valuation into a binary string at a state*[6].

$L(1.01) = 000\ 000,\ L(1.02) = 100\ 000,\ L(1.03) = 010\ 000,\ L(1.04) = 010\ 100,$
$L(1.05) = 010\ 010,\ L(1.06) = 010\ 110,\ L(1.07) = 010\ 001,\ L(1.08) = 110\ 001,$
$L(1.09) = 000\ 001,\ L(1.10) = 100\ 001,\ L(1.11) = 010\ 001.$

Figure 5.11: Reduced label space for local model.

We also reduce the variable statements in the ACTL formula to match the label space. Here, each integer value in the property represents the index of a propositional atoms valuation at a state. In other words, in all state valuation strings, the valuation on $s.sentId_1$ is at index 0, $s.sentId_2$ is at index 1, $s.sentId_4$ is at index 2, $r.receivedId_1$ is at index 3, $r.receivedId_2$ is at index 4, and $r.receivedId_4$ is at index 5. Valuation index values will be referenced as an encoded short hand, as referencing the full propositional atom labels makes the overall formula structure less legible. Using this form of valuation index encoding for properties is used in implementation, as it is faster to map a property atom index to a valuation at a state.

$$\begin{aligned}\mathrm{AG}(&(0 \vee 1 \vee 2) \vee \mathrm{AF}(\neg 3 \wedge \neg 4 \wedge \neg 5) \wedge (\neg 0 \vee 1 \vee 2) \vee \mathrm{AF}(3 \wedge \neg 4 \wedge \neg 5) \wedge \\ &(0 \vee \neg 1 \vee 2) \vee \mathrm{AF}(\neg 3 \wedge 4 \wedge \neg 5) \wedge (\neg 0 \vee \neg 1 \vee 2) \vee \mathrm{AF}(3 \wedge 4 \wedge \neg 5) \wedge \\ &(0 \vee 1 \vee \neg 2) \vee \mathrm{AF}(\neg 3 \wedge \neg 4 \wedge 5))\end{aligned}$$

[6]Valuations are broken into groups of three bits in Figure 5.11 to show the representation of variables *s.sentId* and *r.receivedId*.

As an example we update the state 1.05 by label $r.receivedId = 0$, where 1.05 has the label function $L(1.05) = \{s.sentId_2,\ r.receivedId_2\}$. In the system representation $r.receivedId = 0$ is equivalent to $\neg 3 \wedge \neg 4 \wedge \neg 5$. To satisfy this propositional atom and maintain single value variable consistency, the update $\mathcal{U} = \{(1.05, -, 4)\}$ will satisfy the conditions.

In usage, the returned counterexample with window size five can be visualised using the counterexample transition graph seen in Figure 5.10[7]. Comparing the given states in the Kripke model against the clauses of the model property we can see where each state violates the property.

$1.01 \ldots 1.05$	$\models \phi$
$1.06 \ldots 1.07$	$\not\models (\neg(s.sendId = 2)) \vee \text{AF}(r.receivedId = 2)$
1.08	$\not\models (\neg(s.sendId = 3)) \vee \text{AF}(r.receivedId = 3)$
1.09	$\not\models (\neg(s.sendId = 0)) \vee \text{AF}(r.receivedId = 0)$
1.10	$\not\models (\neg(s.sendId = 1)) \vee \text{AF}(r.receivedId = 1)$
1.11	$\not\models (\neg(s.sendId = 2)) \vee \text{AF}(r.receivedId = 2)$

Figure 5.12: Counterexample states satisfying clauses of the property.

We now analyse the counterexample structure and determine which states do not satisfy which sections of the property given and get an intuition of how the modification can be performed to satisfy the property. We can see that states *1.01* through *1.05* satisfy the property, as every value in the domain of *r.receivedId* eventually occurs in all future states. We can see that states 1.06, 1.07 and 1.11 do not satisfy $(\neg(s.sendId = 2)) \vee \text{AF}(r.receivedId = 2)$ as $s.sendId = 2$ is true at each of the listed states and $r.receivedId = 2$ never becomes true for the cases of states 1.08, 1.09 and 1.10. This trend continues as we find they do not satisfy the sub-property for window values 3, 0 and 1 respectively. To satisfy these sub-properties we can see that we will rather have to remove $s.sentId = n$ from the current state or at some future state by $r.receivedId = n$. As was discussed in Chapter 2, the intuition behind update is that modification which occurs at lower depths of the tree-like model will be favoured over higher depth modifications. We can see that updates which favour the subproperty $\text{AF}(r.receivedId = n)$ will be enacted to satisfy the property.

[7]We show the simplified counterexample, states which held no change between transitions were abstracted out to illustrate the violation, the original counterexample contained nineteen states and three variables. This can be seen in Appendix C.2.

With this analysis complete we apply an update session using $Update_c$, such that the given local models satisfies the properties with their allocated window size. The process is timed to establish efficiency of the update tool against theorised complexity.

5.2.6 Deriving Update

For each operation type encountered in the property, the semantics from Definition 2.1 determines how the model can be manipulated to satisfy the property. For the property in the example the ACTL tokens are AG, AF, $\vee, \wedge$ and $\neg$[8]. Based on the property operation and state passed as argument the update will be routed to the method created to enact the update as described in Chapter 3.

In $Update_c$ a high level update will be performed for the AG operator using the method $Update_{AG}$[9]. The nested disjunctive clauses will be evaluated with the $Update_{\vee}$ member function, evaluating its lower level conjunctive terms to determine which is the minimal update. In $Update_{AG}$ each state in the path is checked to determine what the possible updates are that will satisfy the current formula at a given state. As the subformula translates to $\neg s.sentId_n \vee$ AF$(r.receivedId_n)$[10] we have three operations to process for each state. Disjunction goes first with $Update_{\vee}$ and returns the minimal result to the calling method (*i.e.* $Update_{AG}$). $Update_{\wedge}$ will evaluate terms of atoms and return the set of updates which are required to be performed to satisfy the given state.

These results include applying $\neg sentId_n$ and AF$(receivedId_n)$ to each appropriate state using the respective member functions $Update_p$ and $Update_{AF}$. In the prior case, we determine the result of substituting the state with another not satisfying the label $sentId_n$ and satisfy some other domain value n for $sentId$. This will result in two cases. Rather the state satisfies the negative valuation on the label and returns the empty update set $\mathcal{U} = \emptyset$, or doesn't satisfy the label and the update set is returned as applying some other valuation from the domain at the state. In the former case, we apply the update for the universal future token AF$(r.receivedId_n)$. As characterised earlier, we traverse the model and apply the update to each of the

[8]For further information on semantic satisfaction of CTL temporal operators see [66].

[9]The update is high level in the sense that it is being called from the root of the property formula and execution requires filtering down to each node of the formula tree.

[10]In the case of $n = 5$, $s.sentId$ reduces to 0, 1 and 2 and $r.receivedId$ 3, 4, and 5.

lowest states in each branch of the tree-like model. In this case the only branch available will terminate with a SCC. In this case AF semantics dictate that SCC states are depth equivalent and the first entry state into the SCC is sufficient for update. This will return the update set which modifies $r.receivedId_x$ at the first SCC entry point state.

As only one valuation can be applied to a variable of a certain domain at one state, consistency of updates becomes an issue when multiple updates need to occur with different domain valuations for the same variable. This is especially the case in SCCs where characterisations indicate that updating the first entry point state is the best approach. As states in a SCC are often depth equivalent we can maintain variable valuation consistency by staggering updates such that update is effected and consistency in valuation is maintained at a state level. This issue can also be solved by referring to persistence properties and extending paths to satisfy the necessary property.

Applying AG semantics to the counterexample, it is found that states $1.06 \ldots 1.11$ do not satisfy some subformula of the overall desired property. Applying the update process to the counterexample given, we find the following possible updates are applicable for each state.

Figure 5.13 represents each update option for $Update_{AG}$. We can see that at each state the option exists to satisfy the state by substituting it with another which does not satisfy the state value given for $s.sentId$ and some other value from the domain to replace the previous value. Another possibility exists for update which involves satisfying AF($r.receivedId_n$) at some state in the SCC, and substituting the previous valuation on the variable with another, based on the notion that SCC states are depth equivalent. Looking at these possible updates we compare them with the *minChange* algorithm discussed in Subsection 3.2.6 of Chapter 3 and compare each update.

Looking at the possibilities, we note that the first, updating each state by its first update $s.sentId_n$ gives a relatively low weighting, as many of the depth values for the updates occur higher in the model than the updates involving AF($r.receivedId_n$) were used. As states in a SCC are considered equal based on depth, if we considered only the updates which occur inside the SCC (the deepest region) there would be

$$\textbf{1.06}: \quad \mathcal{U} = \{\{(1.06,+,0) \vee (1.06,-,1) \vee (1.06,+,2) \vee (1.06,-,3)\} \vee \{(1.07,+,4) \wedge (1.07,-,5)\} \vee \{(1.08,+,4) \wedge (1.08,-,5)\} \vee \{(1.09,+,4) \wedge (1.09,-,5)\} \vee \{(1.10,+,4) \wedge (1.10,-,5)\} \vee \{(1.11,+,4) \wedge (1.11,-,4)\}\};$$

$$\textbf{1.07}: \quad \mathcal{U} = \{\{(1.07,+,0) \vee (1.07,-,1) \vee (1.07,+,2)\} \vee \{(1.07,+,4) \wedge (1.07,-,5)\} \vee \{(1.08,+,4) \wedge (1.08,-,5)\} \vee \{(1.09,+,4) \wedge (1.09,-,5)\} \vee \{(1.10,+,4) \wedge (1.10,-,5)\} \vee \{(1.11,+,4) \wedge (1.11,-,5)\}\};$$

$$\textbf{1.08}: \quad \mathcal{U} = \{\{(1.08,-,0) \vee (1.08,-,1) \vee (1.08,+,2)\} \vee \{(1.08,+,3) \wedge (1.08,+,4) \wedge (1.08,-,5)\} \vee \{(1.09,+,3) \wedge (1.09,+,4) \wedge (1.09,-,5)\} \vee \{(1.10,+,3) \wedge (1.10,+,4) \wedge (1.10,-,5)\} \vee \{(1.11,+,3) \wedge (1.11,+,4) \wedge (1.11,-,5)\}\};$$

$$\textbf{1.09}: \quad \mathcal{U} = \{\{(1.09,+,0) \vee (1.09,+,1) \vee (1.09,+,2)\} \vee (1.08,-,5) \vee (1.09,-,5) \vee (1.10,-,5) \vee (1.11,-,5)\};$$

$$\textbf{1.10}: \quad \mathcal{U} = \{\{(1.10,-,0) \vee (1.10,+,1) \vee (1.10,+,2)\} \{(1.08,+,3) \wedge (1.08,-,5)\} \vee \{(1.09,+,3) \wedge (1.09,-,5)\} \vee \{(1.10,+,3) \wedge (1.10,-,5)\} \vee \{(1.11,+,3) \wedge (1.11,-,5)\}\};$$

$$\textbf{1.11}: \quad \mathcal{U} = \{\{(1.11,+,0) \vee (1.11,-,1) \vee (1.11,+,2)\} \vee \{(1.08,+,4) \wedge (1.08,-,5)\} \vee \{(1.09,+,4) \wedge (1.09,-,5)\} \vee \{(1.10,+,4) \wedge (1.10,-,5)\} \vee \{(1.11,+,4) \wedge (1.11,-,5)\}\};$$

Figure 5.13: Possible types of update on Sliding Window $n = 5$ for property AGϕ, where each tuple represents a state s, a modifier $+/-$ and a propositional atom valuation index.

$!4 \cdot 2^3 = 192$ combinations of updates[11].

Comparing some of the possible updates, we can see one possible update which can satisfy the property is

$$\mathcal{U} = \{(1.11,-,1),\ (1.11,+,0),\ (1.11,+,4),(1.11,-,5),\ (1.09,-,5),\ (1.08,+,3),\ (1.08,+,4),\ (1.08,-,5),\ (1.10,+,3),\ (1.10,-,5)\}.$$

Although this is possible and would satisfy the overall property, it is a subset of

[11]This works with the reasoning that we have the permutations of updates over the four states where there also exists two possible modifications to satisfy three states.

another possible update

$$\begin{aligned}\mathcal{U} = \quad & \{(1.10,+,3),\ (1.10,-,5),\ (1.09,-,5),\ (1.08,+,3),\ (1.08,+,4),\\ & (1.08,-,5),\ (1.11,+,4),\ (1.11,-,5)\}\ .\end{aligned}$$

which would be selected as a better update than the former in *minChange*.

5.2.7 Derived Fix

In applying the update, algorithm $Update_c$ returned finding the most minimal update could be performed by replacing states with states that satisfy the variable *r.receivedId* on future paths by the values *0, 1, 2, 3 and 4*, giving a total update count of 8. The update was represented by the returned update tuple set

$$\begin{aligned}\mathcal{U} = \{&(1.08,-,5),\ (1.09,+,3),(1.09,-,5),\ (1.10,+,4),\ (1.10,-,5),\\ &(1.11,+,3),\ (1.11,+,4),\ (1.11,-,5)\}\end{aligned}$$

It is evident that these eight atomic updates each satisfy multiple states, based on the necessity of AG to have each state satisfy the subformula (see Figure 5.12). Viewing this as a property to satisfy, the modification evaluates to the local model being updated by the formula

$$\begin{aligned}&\mathrm{AF}(r.receivedId_2)\wedge\ \mathrm{AF}(r.receivedId_3)\wedge\\ &\mathrm{AF}(r.receivedId_0)\wedge\ \mathrm{AF}(r.receivedId_1)\wedge\neg s.sentId_4.\end{aligned}$$

Calling the member function $Update_{apply}$ with the model from Figure 5.7 and update set, we derive the local model shown in the transition graph in Figure 5.14.

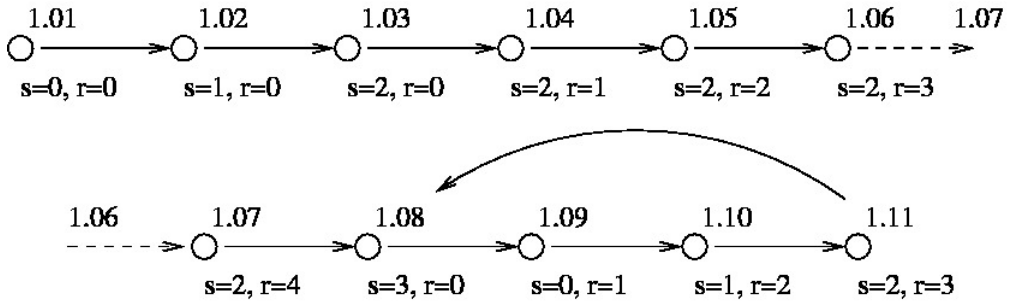


Figure 5.14: Updated local model fix (M', s') for window size $n = 5$.

5.2.8 Results

n	*Formula Atom Count*	*Local Model State Count*	*Total State Count*	*Reachable State Count*	*Process Time (ms)*
2	4	7	8	6	344.00
3	6	13	27	27	1938.00
4	8	12	64	45	5813.00
5	10	19	125	96	22531.00
6	12	31	216	175	94531.00
7	14	32	343	288	278844.00
8	16	33	512	441	1020688.00

Table 5.1: Results of Update on sliding window $n = 2 \ldots 8$.

From Table 5.1, we can see that counterexample size manages to maintain a relatively small size in comparison to the overall reachable state count. Most of the complexity in terms of time is attributable to the large sets of clauses containing AF tokens over AG tokens. As mentioned in Chapter 2, Subsection 2.4.2, calculating minimal updates for both AFϕ and AGϕ where ϕ is a propositional formula can be done in $\mathcal{O}(|R| \cdot |S| \cdot 2^{|var(\phi)|})$ and $\mathcal{O}(|S| \cdot 2^{|var(\phi)|})$ respectfully. In this case study nesting these temporal operators causes computation time to explode quickly for even relatively small values of n, where n is window size. As n is grows, the propositional variable count grows in time $2n$. This implies that for each state update, it will take time $\mathcal{O}(2^{2n})$. For window size 8, for example, it will take $\mathcal{O}(2^{16})$ for each state update in the worst case. Since the counterexample size remains small comparing to the whole model size, the prototype tool is able to perform effective updates and assist to derive actual fixes.

5.3 Case Study 3: The Mutual Exclusion Program

In this section, we provide some information one particular case study - the well known mutual exclusion program, and show how our approach can help to find a proper system modification in the SPIN model checker[12]. This case study was conducted as a preliminary experiment into the application of constraint automata to limit and guide possible modifications in the SPIN environment.

Consider the concurrent program encoded in SPIN in Algorithm 5.1. The program consists of processes PA() and PB(), which share two common boolean variables x and y. To ensure mutual exclusion of the assignments to x and y, some control variables *flag* and *turn*, are introduced. We then declare two critical sections for each process, one for the assignments to x (*line 13 in PA() and PB(), and lines 45/54 in PB()*), and another critical region for the assignments to y in PA() (*lines 23 in PA() and 52 in PB(), respectively*).

It can be seen that in PB(), the critical section for y is nested into critical section for x. Each variable *flagiV* (i = 1, 2 and V = A, B) indicates a request for process PV() to enter critical section i, and *turniB* dictates whether such a request by process PB(), in the scenario where there are simultaneous requests, should be granted.

The specification is formalized in an ACTL formula: $\phi \equiv \mathrm{AG}(\neg(ta \wedge (tb \vee tc)))$. This describes that the program satisfies mutual exclusion for assignments to variables x and y, respectively. This has the effect of eliminating cases where PA() executes line 13 while PB() attempts to execute lines 45 or 54.

As mentioned by Buccafurri in [10], this program contains approximately 10^5 states. We apply the SPIN model checker to check whether this program satisfies property ϕ. With SPIN optimisation, the program still contains 1800 states during the checking process. After the SPIN model checking session, it reports that the

[12]This study was original presented at the Knowledge-Based and Engineering Systems conference 2010[71].

Algorithm 5.1: An example of mutual exclusion - SPIN source code.

```
01:  bool flag1A, flag2A;
02:  bool turn1B, turn2B;
03:  bool flag1B, flag2B;
04:  bool x, y;
05:  bool ta, tb, tc, td;
06:  proctype PA() {
07:    do
08:    :: flag1A = true;
09:    turn1B = false;
10:      do
11:      ::if
12:        :: !flag1B || !turn1B →
13:        atom{x=x && y; ta = true;}
14:        ta = false;
15:        flag1A = false;
16:        if
17:        :: turn1B →
18:        flag2A = true;
19:        turn2B = true;
20:        do
21:        ::if
22:          ::!flag2B || !turn2B →
23:          y = false;
24:          tc = false;
25:          flag2A = false;
26:          break;
27:          ::else;
28:        fi;
29:      od;
30:      ::else;
31:      fi;
32:      break;
33:      ::else;
34:    fi;
35:    od;
36:    od;
37:  }
38:  proctype PB() {
39:    do
40:      :: flag1B = true;
41:      turn1B = false;
42:      do
43:      if
44:        :: !flag1A || !turn1B →
45:        atom{x=x && y;tc = true;}
46:        tb = false;
47:        flag2B = true;
48:        turn2B = false;
49:        do
50:          :: if
51:          :: !flag2A || turn2B →
52:          y = !y
53:          td = false;
54:          atom{x = x||y;tb = true;}
55:          tb = false;
56:          flag2B = false;
57:          flag1B = false;
58:          break;
59:          ::else;
60:          fi;
61:        od;
62:        break;
63:        ::else;
64:        fi;
65:      od;
66:    od;
67:  }
68:
69:  int {
70:    run PA();
71:    run PB();
72:  }
73:
```

property ϕ does not hold for the entered program and returns a linear tree-like counterexample explaining the fault. This counterexample contains 22 states, as shown in Figure 5.15. In Figure 5.16 we have actions mapping to transitions in the counterexample.

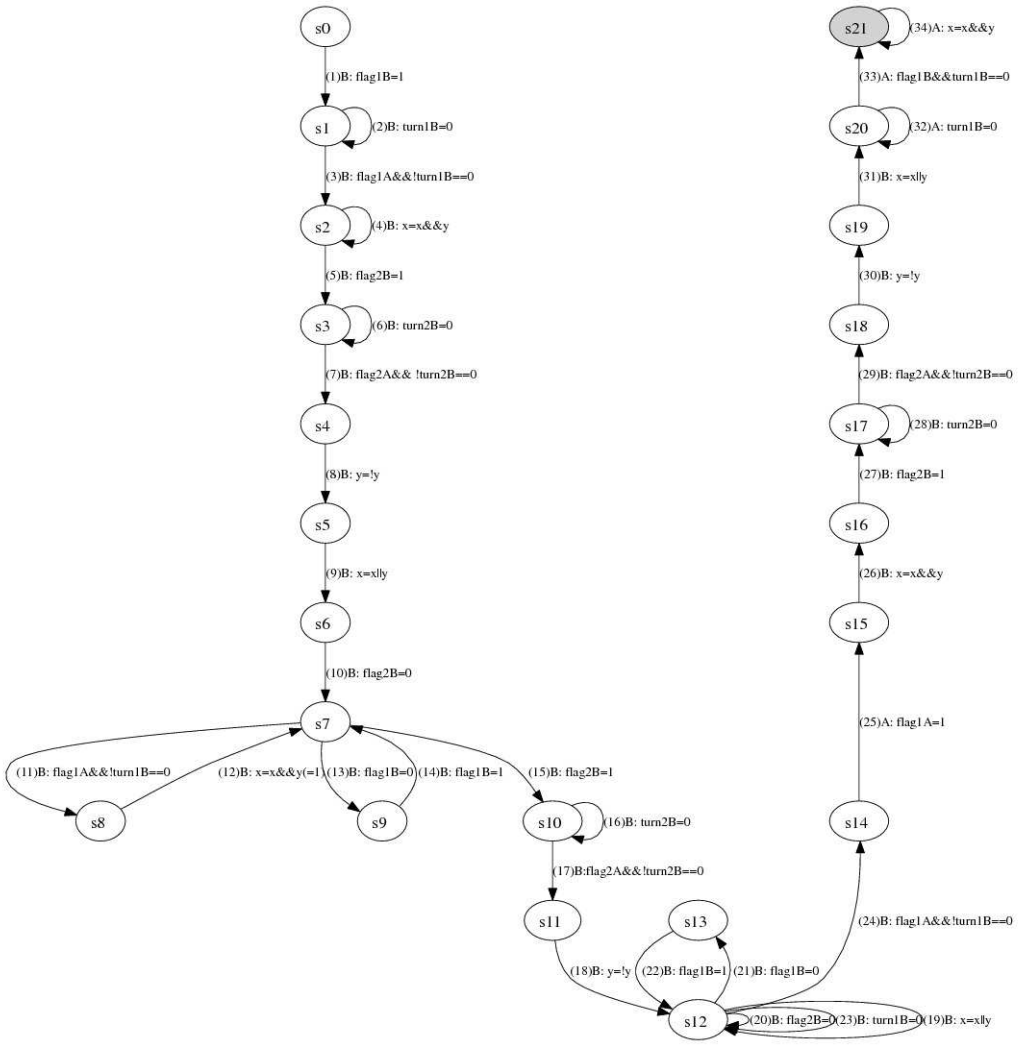


Figure 5.15: A counterexample for AG($\neg(ta \wedge (tb \vee tc))$.

With the counterexample extracted, we determine a means for applying local model update. First, we construct relevant constraint automata for this program, as discussed in Section 5.3 of Chapter 4. In this case study, the particular variable constraint automaton shown in Figure 5.17 will be directly embedded into the update process.

In Figure 5.17, "$*1$" indicates any statement in process PA() except statement

01 (s_0, s_1) B: $flag1B = 1$	*18* (s_{11}, s_{12}) B: $y =!y$
02 (s_1, s_1) B: $turn1B = 0$	*19* (s_{12}, s_{12}) B: $x = x\|\|y$
03 (s_1, s_2) B: $flag1A\&\&!turn1B == 0$	*20* (s_{12}, s_{12}) B: $flag2B = 0$
04 (s_2, s_2) B: $x = x\&\&y$	*21* (s_{12}, s_{13}) B: $flag1B = 0$
05 (s_2, s_3) B: $flag2B = 1$	*22* (s_{13}, s_{12}) B: $flag1B = 1$
06 (s_3, s_3) B: $turn2B = 0$	*23* (s_{12}, s_{12}) B: $turn1B = 0$
07 (s_3, s_4) B: $flag2A\&\&!turn2B == 0$	*24* (s_{12}, s_{14}) B: $flag1A\&\&!turn1B == 0$
08 (s_4, s_5) B: $y =!y$	*25* (s_{14}, s_{15}) A: $flag1A = 1$
09 (s_5, s_6) B: $x = x\|\|y$	*26* (s_{15}, s_{16}) B: $x = x\&\&y$
10 (s_6, s_7) B: $flag2B = 0$	*27* (s_{16}, s_{17}) B: $flag2B = 1$
11 (s_7, s_8) B: $flag1A\&\&!turn1B == 0$	*28* (s_{17}, s_{17}) B: $turn2B = 0$
12 (s_8, s_7) B: $x = x\&\&y(= 1)$	*29* (s_{17}, s_{18}) B: $flag2A\&\&!turn2B == 0$
13 (s_7, s_9) B: $flag1B = 0$	*30* (s_{18}, s_{19}) B: $y =!y$
14 (s_9, s_7) B: $flag1B = 1$	*31* (s_{19}, s_{20}) B: $x = x\|\|y$
15 (s_7, s_10) B: $flag2B = 1$	*32* (s_{20}, s_{20}) A: $turn1B = 0$
16 (s_{10}, s_{10}) B: $turn2B = 0$	*33* (s_{20}, s_{21}) A: $flag1B\&\&!turn1B == 0$
17 (s_{11}, s_{11}) B: $flag2A\&\&!turn2B == 0$	*34* (s_{21}, s_{22}) A: $x = x\&\&y$

Figure 5.16: Action map for mutual exclusion counterexample.

$A : x = x\&\&y$, and "$*2$" indicates any statement in process PB() except statements $B : x = x\&\&y$ and $B : x = x\|y$. Intuitively, this variable constraint automaton represents the constraints between variables ta and tb with respect to the various actions (statements) given in the program.

Considering the counterexample shown in Figure 5.15, we observe that state s_{21} describes the violation on property ϕ, where $L(s_{21}) = \{ta = 1, tb = 1, tc = 0, flag1A = 1, ...\}$. We can see that the transition from s_{20} to s_{21} in Figure 5.15 corresponds to the automaton states s_2 and s_3 in Figure 5.17 respectively.

By applying our tree-like local model update with the associated constraint automaton (Definition 2.10), the counterexample will be minimally updated to satisfy ϕ: state s_{21} can be updated to either (1) s'_{21}: $L(s'_{21}) = \{ta = 0, tb = 1, tc = 0, flag1A = 1, ...\}$, or (2) s''_{21}: $L(s''_{21}) = \{ta = 1, tb = 0, tc = 0, flag1A = 1, ...\}$, while all other states in the local model will remain unchanged.

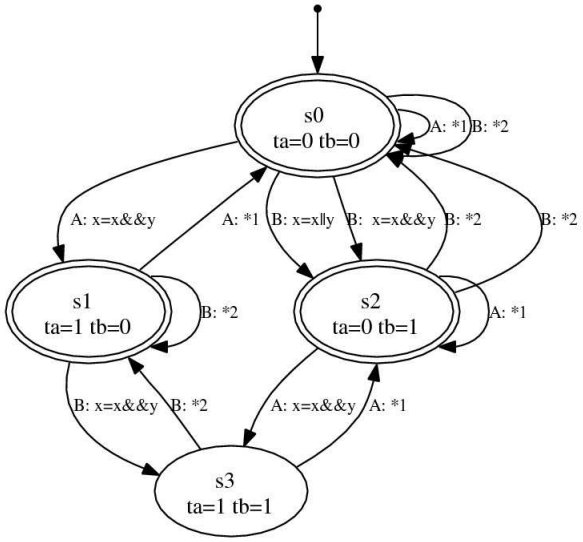


Figure 5.17: The variable constraint automaton for *ta* and *tb*.

This update suggests that one possible modification for the original program in Algorithm 5.1 is to change *line 9* in PA() from "turn1B = false;" to "turn1B = true;". Applying further SPIN model checking using the revised mutual exclusion program will confirm that this result from the local model update process is a final correction to the original program.

5.4 Summary

In this chapter we have given three significant case studies demonstrating the usage of the local model update process and how it can be used to determine fixes for local models derived from counterexamples of real world model checking programs. As can be seen from these examples, update becomes computationally expensive as property and reachable state size grows, making scaling difficult. Further to this, any aspects of the model not represented with temporal properties can be lost when applying an update.

Note that although local model update was shown for single counterexample traces the approach works for crafted tree-like local models. However, multiple counterexamples need to be parsed and typically NuSMV only returns single paths leading to a violation of the property. In the following chapter we review the content presented in this thesis and provide possible direction for future analysis in this field of research.

Chapter 6

Conclusion

In this chapter, we give a final summation of concepts in this dissertation and discuss the approach limitations. We then lead to further possible research in the field of model-based modification and repair.

6.1 Research Summary

As emphasised earlier, model checking is a sound tool for the verification of hardware and software models. Saying this, it has long been known not to be directly applicable to systems modification and previous methods of system repair and model update are infeasible for industrial scale application. Techniques previously devised take the model's entire state space to apply update and previous implementations of the approach were taylored to specific case studies. Furthermore, ordering rules such as closeness, admissible update and maximal reachability still allow much of the effect of the model explosion problem to occur. The research conducted in this thesis has extended this work by providing a means of updating a localised region, describing the violation on the model using model selection semantics that encourage minimal change and maximal reachability in the region and satisfying the necessary temporal property. Also, we note the importance of persisting properties, such that when satisfaction of a new property is required, previous properties still hold after the modification is made. Characteristics of persistence are analysed and shown to be maintained for specific cases of update.

In this thesis we have given a thorough analysis of the theoretical background of ACTL local model update and provided a system prototype which generates candidate fixes from counterexample traces. Further to this we have provided the theory and implementation of constraint compliance with action and variable constraints to derive updates which correspond better to system behaviour which could not be expressed through temporal properties. This addition to model update functionality represents a significant improvement in efficiency and control in the update process, alleviating challenges related to the state and model explosion problems.

This research project is another stepping stone for future researchers in generating an efficient model update compilation tool which can extend from the theoretical and implemented techniques this research thesis. This being said there is much research to be done to increase technique efficiency and theoretical aspects to extend the approach and lend more generality. Detailed in the following section is a summary of research provided in each chapter and an analysis of potential avenues for further study in the field of model update based on the current limitations evident.

Minimal Change with Weak Bisimulation Ordering

Local models are characterised as Kripke structures which hold a tree-like structure, generated from counterexamples derived from model checking sessions with ACTL properties. Local model update is guided by weak bisimulation as a means of determining which modified local model is closest to the original relative to some other local model which satisfies the property. This is done by mapping similarity in the transition structure of a candidate model with the original. In this way two sets of modifications can be compared to determine which is optimal. The semantics of weak bisimulation guided ordering favours model modifications driven to the lower depths of a tree-like model. We found that for each of the temporal operators of ACTL updates occurring on sections of the model, where possible and applicable, updating on lower branches is favoured if possible. This form of ordering criteria was tempered with variable and action constraint automata as a means of directing update towards modifications, which, although may be less desirable from a weak bisimulation ordering standpoint, will comply to necessary constraint automata. Further, a system for tokenising updates with update tuples, is posed. This allows

us to map differences between two models in an efficient way using atomic updates similar to primitive updates given in [101].

Update Characterisations and Complexity

ACTL properties are abstractions of system behaviours which should occur in the underlying model. As the approach is applied to local models generated from counterexamples in checking sessions, we are guaranteed that the local model does not satisfy the property. In performing the updates we give characterisations of updates and provide time complexities given the type of property operations encountered. The complexity of a local model update is proven to be *co-NP-complete*. Persistence properties are shown to allow some properties to be satisfied in a model and persist while being updated by another property. This is in the scenarios where the properties hold no common propositional atoms or the update involves strict extensions.

Algorithms

The algorithms generated to enact ACTL Local model update have been done so guided by the characterisations generated in the earlier theoretical section. Algorithms are expressed as pseudocode and uses a combination of iterative and recursive means to effect path traversal and property satisfaction within the local model.

The algorithm provides a central function which routes control to its subordinate methods, which in turn recursively call the central method or, if passed a propositional atom or some heuristic is designed for satisfying the current formula, will return an update tuple set which suggests a state, transition or label to add or remove, based on ACTL formula semantics in Definition 2.1. When the most minimal update has been determined it is recursively returned back through the function stack. This update set can be applied to the model to effect property satisfaction and the algorithm may terminate. With this, SMV counterexamples can be reduced to the corresponding Kripke structures and the described algorithms can be used to modify it to satisfy the property in a minimal way.

Prototype

Given the generated local model algorithms and characterisations, we have developed a prototype *l-Up* for updating tree-like models derived from model checking counterexamples. The prototype works as a back-end to the model checking tool NuSMV, allowing analysis and parsing of counterexamples into an associated tree-like model transition system with state labels. The prototype uses the previously described algorithms to effect satisfaction of the properties and to maintain satisfaction of properties which still hold for the local model. The *l-Up* project provides 4 packages for research in local model update, including *pySMV* for extracting SMV program and counterexample details, *pyFormula* for parsing, lexical analysis and manipulation of ACTL formulas, *pyModels* for providing core local model update functionality and local model checking, and *pyAutomata* for constraint automata analysis.

Case Studies

This approach has been demonstrated on three robust case studies, two expressed as SMV input files, a semaphore sharing model and the Sliding Window Protocol modelled with increasing window size, and the mutual exclusion model in the SPIN environment. With each of these examples a minimal update was derived which satisfied the local model in question in a computationally feasible time and model size.

The implementation presented demonstrates the research goals of this dissertation: local model update is shown to be a viable approach in respect to complexity in Chapter 2. The final example demonstrated the application of constraint automata to an instance of a mutual exclusion model in the SPIN environment, showing how variable constraints can be used to devise applicable updates in a process locking scenario.

Domain Constraints

Extending the update approach, we included constraint automata compliance theory to allow system designers to specify more fine grained control of update behaviour and to ascertain what types of modifications are admissible, other than just specifying the universal temporal specification which needs to be satisfied. Action and variable constraint automaton were considered for expressing various conditions in the model otherwise inexpressible in ACTL. Definitions for constraint compliance were given, characterisations and complexity results were ascertained and algorithms were generated describing how this is implemented in the local model update prototype.

To show the efficacy of local model update as extended by constraint automata we have included a case study of the mutual exclusion model, using variable constraint automata to dictate correct behaviour in a counterexample towards satisfying a user defined property.

6.2 Future Research

6.2.1 Model Checking and Tree-like Models

To be able to successfully effect global satisfaction of some system model found to not satisfy an ACTL property we require model checking tools which can generate tree-like models. At the current state tree-like models are generated by deriving linear counterexamples from NuSMV and mapping the same states to merge the transition systems in such a way as to maintain the tree-like structure. Having tree-like model generation be a standard feature for model checkers would further help automate the process. In [27] Clarke et al. proposes symbolic algorithms for computing tree-like models in the SMV model checking tool, however no implementation or smv extension has been proposed as of yet.

6.2.2 Reintegration of the Updated Local Model

The approach given in this thesis proposes a method for devising minimal candidate fixes for localised tree-like models derived from counterexamples. The next logical step is to determine a method for reintegration of the local model back into the global model such that the original model satisfies the required property and any other defined properties.

Inherent in this process is the requirement of mapping a repaired tree-like model to behaviour in some SMV file, representable by a Kripke structure. This involves multiple steps. Firstly, we would need to develop a method for translating the local model repair back into an explicit Kripke structure such that the global Kripke structure also satisfies the property (*i.e.* a global fix). The next step would be to translate this approach back to the original model specification file such that the new smv file satisfies the temporal property. This would be the next step in developing a universal ACTL local model updater.

6.2.3 Automatic Generation of Constraint Automata

Constraint automata are a useful tool for guiding update modifications. However, it is the job of the developer to analyse the system design, model and construct the automata necessary to guide update. A fruitful research direction would be to investigate the automatic generation of these structures through analysis of model description files. Algorithms could be used to determine related variable domains and process action interaction such that the update process can have a minimal footprint on necessary system structure while still guaranteeing the temporal property is satisfied in the model.

6.2.4 Model Checker Parsers and Extraction Tools

Many of the approaches shown in this thesis relied on being able to parse the input specification and output reports of the NuSMV suite, and with further development on the parsers and extraction tools many other applications can be generated. This includes computation of tree-like models, generation of constraint automata and steps toward mapping the derived modification back into a model checking specification file. Integrated tools which provide this functionality would be useful for future research into generating a universal update tool.

6.3 Summary

We have provided a theoretical and prototypical foundation for the approach of applying update to local regions of a system model represented as tree-like Kripke structures. To emphasise, this work provides a foundation for the generation of a robust universal model update tool which returns minimal candidate modifications. There exist many avenues for continuing research at later stages leading to this goal. Continuing from this work, we are confident that this research solves many of the challenges posed on the path to creating a universal model update tool.

A. Read Me File for the ACTL Tree-like Local Model Update System

l-Up is a modular software package written in Python that can be used as an API to be built upon by future researchers looking to develop and extend the technique of localised treelike model update. The package contains a library of Python packages and modules for the purposes of model checking, counterexample interpretation and parsing, tree-like structure building and parsing, ACTL formula tokenisation and interpretation.

Also included is example programs used in this thesis to test the validity of the local model update system. This includes:

- Gigamax coherence protocol for variable parsing;
- Semaphore sharing;
- Sliding window protocol with varying window size.

The Tree-like local model update suite can be obtained from attached CD disk or the following URL, and comes compressed for windows as *.zip* respectfully.

http://scm.uws.edu.au/∼mkelly/l-upProject.html

A.1 Windows

The development package *l-Up* was created using the *Python(x, y)*. *Python(x, y)* a scientific-oriented Python distribution aimed at providing engineering development software for numerical computations, data analysis and data visualization. Included with *Python(x, y)* is the *Eclipse* development environment with the Python project development tool *PyDev*.

To test the case studies given in this thesis in *Eclipse* or to analyse the *l-Up* project, the reader needs to import the *l-Up* project into the *Eclipse* workspace. To import the *l-Up* package right-click inside the *PyDev Package Explorer* and select *Import....* In the Import window double-click *General* $\rightarrow$ *Existing Projects into Workspace*. From here select *Browse* next to the *Select root directory* title and indicate the root directory of the *l-Up* directory.

To execute the scenarios presented in the case studies select the *experiments* package in *src* and select one of the modules. To run each case select $F9$ to execute the program.

As *l-Up* has been scripted in the Python scripting language it is accessible on any operating system with the Python 2.6 interpreting libraries installed. *NuSMV 2.4.3* is included with the package for the purpose of deriving counterexamples. The latest version NuSMV 2.5.3 is currently untested with this framework.

Further, for formula parsing the *PLY* tools *yacc* and *lex* are necessary. To install PLY download the package from *http://www.dabeaz.com/ply/*. Go to command prompt and navigate to the PLY package folder, from here, type

python setup.py install.

Python will now allow reference to *yacc* and *lex* in the *eclipse* environment.

B. Gigamax Cache Coherence Model Program

Algorithm B.1 Processor module.

```
01:  MODULE processor(CMD, REPLY-OWNED, REPLY-WAITING, REPLY-STALL)
02:  ISA bus-device;
03:  ISA cache-device;
04:  ASSIGN
05:      cmd :=
06:          case
07:              master & state = invalid : {read-shared, read-owned};
08:              master & state = shared : read-owned;
09:              master & state = owned & snoop = owned : write-resp-invalid;
10:              master & state = owned & snoop = shared: write-resp-shared;
11:              master & state = owned & snoop = invalid: write-invalid;
12:              TRUE: idle;
13:          esac;
```

Algorithm B.2.1 Cache-device module.

```
01: MODULE cache-device
02: VAR
03:     state : {invalid, shared, owned};
04: DEFINE
05:     readable := ((state = shared) | (state = owned)) & !waiting;
06:     writable := (state = owned) & !waiting;
07: ASSIGN
08:     init(state) := invalid;
09:     next(state) :=
10:       case
11:         abort: state;
12:         master :
13:           case
14:             CMD = read-shared : shared;
15:             CMD = read-owned : owned;
16:             CMD = write-invalid : invalid;
17:             CMD = write-resp-invalid : invalid;
18:             CMD = write-shared : shared;
19:             CMD = write-resp-shared : shared;
20:             TRUE : state;
21:           esac;
22:         !master & state = shared & (CMD = read-owned | CMD = invalidate) :
23:           invalid;
24:         state = shared : {shared, invalid};
25:         TRUE : state;
26:       esac;
27: DEFINE
28:     reply-owned := !master & state = owned;
29: VAR
30:     snoop : {invalid, owned, shared};
31: ASSIGN
32:     init(snoop) := invalid;
33:     next(snoop) :=
34:         case
```

Algorithm B.2.2 Cache-device Module cont.

```
35:             abort : snoop;
36:             !master & state = owned & CMD = read-shared: snoop;
37:             !master & state = owned & CMD = read-shared: owned;
38:             master & CMD = write-resp-invalid : invalid;
39:             master & CMD = write-resp-shared : invalid;
40:             TRUE : snoop;
41:         esac;
```

Algorithm B.3 Bus-device module.

```
01: MODULE bus-device
02: VAR
03:     master : boolean;
04:     cmd : {idle, read-shared, read-owned, write-invalid, write-shared,
05:         ... write-resp-invalid, write-resp-shared, invalidate, response};
06:     waiting : boolean;
07:     reply-stall: boolean;
08: ASSIGN
09:     init(waiting) := FALSE;
10:     next(waiting) :=
11:         case
12:         abort : waiting;
13:         master & CMD = read-shared : TRUE;
14:         master & CMD = read-owned : TRUE;
15:         !master & CMD = response : FALSE;
16:         !master & CMD = write-resp-invalid : FALSE;
17:         !master & CMD = write-resp-shared : FALSE;
18:         TRUE : waiting;
19:     esac;
20: DEFINE
21:     reply-waiting := !master & waiting;
22:     abort := REPLY-STALL
        ... | ((CMD = read-shared | CMD = read-owned) & REPLY-WAITING);
```

Algorithm B.4 Memory module.

```
01: MODULE memory(CMD, REPLY-OWNED, REPLY-WAITING, REPLY-STALL)
02: VAR
03:     master : boolean;
04:     cmd : {idle, read-shared, read-owned, write-invalid, write-shared,
05:         ... write-resp-invalid, write-resp-shared, invalidate, response};
06:     busy : boolean;
07:     reply-stall: boolean;
08: DEFINE
09:     reply-owned := FALSE;
10:     reply-waiting := FALSE;
11:     abort := REPLY-STALL | (CMD = read-shared | CMD = read-owned) &
            ... REPLY-WAITING | (CMD = read-shared | CMD = read-owned)
            ... & REPLY-OWNED;
12: ASSIGN
13:     init(busy) := FALSE;
14:     next(busy) :=
15:         case
16:             abort : busy;
17:             master & CMD = response: FALSE;
18:             !master & (CMD = read-owned | CMD = read-shared): TRUE;
19:             TRUE: busy;
20:         esac;
21:     cmd :=
22:         case
23:             master & busy : {response, idle};
24:             TRUE: idle;
25:         esac;
26:     reply-stall :=
27:         case
28:             busy & (CMD = read-shared | CMD = read-owned
29:                 | CMD = write-invalid | CMD = write-shared
30:                 |CMD = write-resp-invalid|CMD = write-resp-shared):TRUE
31:             TRUE: {FALSE, TRUE};
32:         esac;
```

Algorithm B.5 Main Module

```
01:  MODULE main
02:  VAR
03:      cmd : {idle, read-shared, read-owned, write-invalid, write-shared,
04:          ... write-resp-invalid, write-resp-shared, invalidate, response};
05:      p0 : processor(CMD, REPLY-OWNED, REPLY-WAITING, REPLY-STALL);
06:      p1 : processor(CMD, REPLY-OWNED, REPLY-WAITING, REPLY-STALL);
07:      p2 : processor(CMD, REPLY-OWNED, REPLY-WAITING, REPLY-STALL);
08:      m : memory(CMD, REPLY-OWNED, REPLY-WAITING, REPLY-STALL);
09:  DEFINE
10:      REPLY-OWNED := p0.reply-owned | p1.reply-owned | p2.reply-owned;
11:      REPLY-WAITING := p0.reply-waiting | p1.reply-waiting | p2.reply-waiting;
12:      REPLY-STALL := p0.reply-stall | p1.reply-stall | p2.reply-stall | m.reply-stall;
13:  ASSIGN
14       CMD :=
15:          case
16:              p1.cmd = idle & p2.cmd = idle & m.cmd = idle : p0.cmd;
17:              p0.cmd = idle & p2.cmd = idle & m.cmd = idle : p1.cmd;
18:              p0.cmd = idle & p1.cmd = idle & m.cmd = idle : p2.cmd;
19:              p0.cmd = idle & p1.cmd = idle & p2.cmd = idle : m.cmd;
20:              TRUE: {idle, read-shared, read-owned, write-invalid, write-shared,
21:                  ... write-resp-invalid, write-resp-shared, invalidate, response};
22:          esac;
23:  ASSIGN
24:      p0.master := {FALSE, TRUE}
25:      p1.master :=
26:          case
27:              p0.master : FALSE;
28:              TRUE : {FALSE, TRUE};
29:          esac;
30:      p2.master :=
31:          case
32:              p0.master | p1.master: FALSE;
33:              TRUE : {FALSE, TRUE};
34:          esac;
35:      m.master :=
36:          case
37:              p0.master | p1.master | p2.master: FALSE;
38:              TRUE : {FALSE, TRUE};
39:          esac;
40:  SPEC AG EF (p0.readable)
41:
42:  SPEC AG EF (p0.writable)
43:
44:  SPEC AG ! (p0.writable & p1.writable)
```

−− specification AG (EF p0.readable) is true
−− specification AG (EF p0.writable) is true
−− specification AG !(p0.writable & p1.writable) is false
−− as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
system diameter: 8
reachable states: 8872 ($2^{13.115}$) out of $1.76319e+011$ ($2^{37.3594}$)

01: − > **State**: 1.1 < −
02: CMD = read-shared
03: p0.master = 1
04: p0.cmd = read-shared
05: p0.waiting = 0
06: p0.reply-stall = 0
07: p0.state = invalid
08: p0.snoop = invalid
09: p1.master = 0
10: p1.cmd = idle
11: p1.waiting = 0
12: s.sentId = 1
13: p1.reply-stall = 0
14: p1.state = invalid
15: p1.snoop = invalid
16: p2.master = 0
17: p2.cmd = idle
18: p2.waiting = 0
19: p2.reply-stall = 0
20: p2.state = invalid
21: p2.snoop = invalid
22: m.master = 0
23: m.cmd = idle
24: m.busy = 0
25: m.reply-stall = 0
26: s.running = 0
27: REPLY-STALL = 0
28: REPLY-WAITING = 0
29: REPLY-OWNED = 0
30: p0.abort = 0
31: p0.reply-waiting = 0
32: p0.writable = 0
33: p0.readable = 0
34: p0.reply-owned = 0
35: p1.abort = 0
36: p1.reply-waiting = 0
37: p1.writable = 0
38: p1.readable = 0
39: p1.reply-owned = 0
40: p2.abort = 0
41: p2.reply-waiting = 0
42: p2.writable = 0
43: p2.reply-owned = 0
44: m.abort = 0
45: m.reply-waiting = 0
46: m.reply-owned = 0
47: − > **Input**: 1.2 < −
48: − > **State**: 1.2 < −
49: CMD = response
50: p0.master = 0
51: p0.cmd = idle
52: p0.waiting = 1
53: p0.state = shared
54: m.master = 1
55: m.cmd = response
56: m.busy = 1
57: REPLY-WAITING = 1
58: p0.reply-waiting = 1
59: − > **Input**: 1.3 < −
60: − > **State**: 1.3 < −
61: CMD = read-shared
62: p0.waiting = 0
63: p1.master = 1
64: p1.cmd = read-shared
65: m.master = 0
66: m.cmd = idle
67: m.busy = 0
68: REPLY-WAITING = 0
69: p0.reply-waiting = 0
70: p0.writable = 1
71: p0.readable = 1
72: − > **Input**: 1.4 < −
73: − > **State**: 1.4 < −
74: CMD = response
75: p1.master = 0
76: p1.cmd = idle
77: p1.waiting = 1
78: p1.state = shared
79: m.master = 1
80: m.cmd = response
81: m.busy = 1
82: REPLY-WAITING = 1
83: p1.reply-waiting = 1
84: − > **Input**: 1.5 < −
85: − > **State**: 1.5 < −
86: CMD = read-shared
87: p1.waiting = 0
88: p2.master = 1
89: p2.cmd = read-shared
90: m.master = 0
91: m.cmd = idle
92: m.busy = 0
93: REPLY-WAITING = 0
94: p1.reply-waiting = 0
95: p1.writable = 1
96: p1.readable = 1

Gigamax program counterexample with inserted error.

C. Counterexamples for Case Studies

∗ ∗ ∗ This is NuSMV 2.4.3 (compiled on Tue May 22 14:08:54 UTC 2007)
∗ ∗ ∗ For more information on NuSMV see <http://nusmv.irst.itc.it>
∗ ∗ ∗ or email to <nusmv-users@irst.itc.it>.
∗ ∗ ∗ Please report bugs to <nusmv@irst.itc.it>.

∗ ∗ ∗ This version of NuSMV is linked to the MiniSat SAT solver.
∗ ∗ ∗ See http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat
∗ ∗ ∗ Copyright (c) 2003-2005, Niklas Een, Niklas Sorensson

−− specification AG !(proc1.state = critical & proc2.state = critical) is true
−− specification AG (proc1.state = entering → AF proc1.state = critical)
is false
−− as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample

Counterexample header for semaphore sharing model.

01: − > **State**: 1.1 < −
02: semaphore = 0
03: proc1.state = idle
04: proc2.state = idle
05: − > **Input**: 1.2 < −
06: _process_selector_ = proc2
07: running = 0
08: proc2.running = 1
09: proc1.running = 0
10: − > **State**: 1.2 < −
11: proc2.state = entering
12: − > **Input**: 1.3 < −
13: _process_selector_ = proc2
14: running = 0
15: proc2.running = 1
16: proc1.running = 0
17: − > **State**: 1.3 < −
18: semaphore = 1
19: proc2.state = critical
20: − > **Input**: 1.4 < −
21: _process_selector_ = proc1
22: running = 0
23: proc2.running = 0
24: proc1.running = 1
25: − > **State**: 1.4 < −
26: proc1.state = entering
27: − > **Input**: 1.5 < −
28: _process_selector_= proc2
29: running = 0
30: proc2.running = 1
31: proc1.running = 0
32: – Loop starts here
33: − > **State**: 1.5 < −
34: proc2.state = critical
35: − > **Input**: 1.6 < −
36: _process_selector_ = proc2
37: running = 0
38: proc2.running = 1
39: proc1.running = 0
40: − > **State**: 1.6 < −
41: semaphore = 0
42: proc2.state = idle
43: − > **Input**: 1.7 < −
44: _process_selector_ = proc2
45: running = 0
46: proc2.running = 1
47: proc1.running = 0
48: − > **State**: 1.7 < −
49: proc2.state = entering
50: − > **Input**: 1.8 < −
51: _process_selector_ = proc2
52: running = 0
53: proc2.running = 1
54: proc1.running = 0
55: − > **State**: 1.8 < −
56: semaphore = 1
57: proc2.state = critical
58: − > **Input**: 1.9 < −
59: _process_selector_ = proc1
60: running = 0
61: proc2.running = 0
62: proc1.running = 1
63: − > **State**: 1.9 < −
64: − > **Input**: 1.10 < −
65: _process_selector_ = proc2
66: running = 0
67: proc2.running = 1
68: proc1.running = 0
69: − > **State**: 1.10 < −
70: proc2.state = exiting
71: *system diameter*: 5
72: *reachable states*:
73: 12 ($2^{3.58496}$) *out of 32* (2^5)

Counterexample for semaphore sharing model with two processes.

∗ ∗ ∗ This is NuSMV 2.4.3 (compiled on Tue May 22 14:08:54 UTC 2007)
∗ ∗ ∗ For more information on NuSMV see <http://nusmv.irst.itc.it>
∗ ∗ ∗ or email to <nusmv-users@irst.itc.it>.
∗ ∗ ∗ Please report bugs to <nusmv@irst.itc.it>.

∗ ∗ ∗ This version of NuSMV is linked to the MiniSat SAT solver.
∗ ∗ ∗ See http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat
∗ ∗ ∗ Copyright (c) 2003-2005, Niklas Een, Niklas Sorensson

−− specification AG $((s.sentId = 0 \rightarrow AFr.receivedId = 0)$ &
$(s.sentId = 1 \rightarrow AFr.receivedId = 1)$ &
$(s.sentId = 2 \rightarrow AFr.receivedId = 2)$ &
$(s.sentId = 3 \rightarrow AFr.receivedId = 3)$ &
$(s.sentId = 4 \rightarrow AFr.receivedId = 4))$
is false
−− as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
⋮
system diameter: 15
reachable states: 96 ($2^{6.58496}$) *out of* 125 ($2^{6.96578}$)

Header for Sliding Window Protocol counterexample of size $n = 5$.

01: − > **State**: 1.1 < −
02: s.sentId = 0
03: r.receivedId = 0
04: a.returnId = 0
05: − > **Input**: 1.2 < −
06: _process_selector_ = s
07: running = 0
08: a.running = 0
09: r.running = 0
10: s.running = 1
11: − > **State**: 1.2 < −
12: s.sentId = 1
13: − > **Input**: 1.3 < −
14: _process_selector_ = s
15: running = 0
16: a.running = 0
17: r.running = 0
18: s.running = 1
19: − > **State**: 1.3 < −
20: s.sentId = 2
21: − > **Input**: 1.4 < −
22: _process_selector_ = a
23: running = 0
24: a.running = 1
25: r.running = 0
26: s.running = 0
27: − > **State**: 1.4 < −
28: a.returnId = 4
29: − > **Input**: 1.5 < −
30: _process_selector_ = r
31: running = 0
32: a.running = 0
33: r.running = 1
34: s.running = 0
35: − > **State**: 1.5 < −
36: r.receivedId = 1
37: − > **Input**: 1.6 < −
38: _process_selector_ = r
39: running = 0
40: a.running = 0
41: r.running = 1
42: s.running = 0
43: − > **State**: 1.6 < −
44: r.receivedId = 2
45: − > **Input**: 1.7 < −
46: _process_selector_ = r
47: running = 0
48: a.running = 0
49: r.running = 1
50: s.running = 0
51: − > **State**: 1.7 < −
52: r.receivedId = 3
53: − > **Input**: 1.8 < −
54: _process_selector_ = a
55: running = 0
56: a.running = 1
57: r.running = 0
58: s.running = 0
59: − > **State**: 1.8 < −
60: − > **Input**: 1.9 < −
61: _process_selector_ = r
62: running = 0
63: a.running = 0
64: r.running = 1
65: s.running = 0
66: − > **State**: 1.9 < −
67: r.receivedId = 4
68: − > **Input**: 1.10 < −
69: _process_selector_ = s
70: running = 0
71: a.running = 0
72: r.running = 0
73: s.running = 1
74: − > **State**: 1.10 < −
75: s.sentId = 3
76: − > Input: 1.11 < −
77: _process_selector_ = a
78: running = 0
79: a.running = 1
80: r.running = 0
81: s.running = 0
82: −− Loop starts here
83: − > **State**: 1.11 < −
84: a.returnId = 3
85: − > **Input**: 1.12 < −
86: _process_selector_ = a
87: running = 0
88: a.running = 1
89: r.running = 0
90: s.running = 0
91: −− Loop starts here
92: − > **State**: 1.12 < −
93: − > **Input**: 1.13 < −
94: _process_selector_ = s
95: running = 0
96: a.running = 0
97: r.running = 0
98: s.running = 1
99: − > **State**: 1.13 < −
100: s.sentId = 0
101: − > **Input**: 1.14 < −
102: _process_selector_ = s
103: running = 0
104: a.running = 0
105: r.running = 0
106: s.running = 1
107: − > **State**: 1.14 < −
108: s.sentId = 1
109: − > **Input**: 1.15 < −
110: _process_selector_= s
111: running = 0
112: a.running = 0
113: r.running = 0
114: s.running = 1
115: − > **State**: 1.15 < −
116: s.sentId = 2
117: − > **Input**: 1.16 < −
118: _process_selector_ = a
119: running = 0
120: a.running = 1
121: r.running = 0
122: s.running = 0
123: − > **State**: 1.16 < −
124: a.returnId = 4
125: − > **Input**: 1.17 < −
126: _process_selector_ = r
127: running = 0
128: a.running = 0
129: r.running = 1
130: s.running = 0
131: − > **State**: 1.17 < −
132: − > **Input**: 1.18 < −
133: _process_selector_ = s
134: running = 0
135: a.running = 0
136: r.running = 0
137: s.running = 1
138: − > **State**: 1.18 < −
139: s.sentId = 3
140: − > **Input**: 1.19 < −
141: _process_selector_ = a
142: running = 0
143: a.running = 1
144: r.running = 0
145: s.running = 0
146: − > **State**: 1.19 < −
147: a.returnId = 3

Counterexample for Sliding Window Protocol of size $n = 5$.

Bibliography

[1] Rajeev Alur, Thomas A. Henzinger, and Pei-Hsin Ho. Automatic symbolic verification of embedded systems. *IEEE Transactions on Software Engineering*, 22:181–201, 1996.

[2] Appcelerator. Pydev, the python project development environment, http://pydev.org/, 2011.

[3] Christel Baier, Joost-Pieter Katoen, and Kim Guldstrand Larsen. *Principles of Model Checking (Representation and Mind Series)*. MIT Press, 2008.

[4] Thomas Ball, Mayur Naik, and Sriram K. Rajamani. From symptom to cause: localizing errors in counterexample traces. *SIGPLAN Not.*, 38:97–105, January 2003.

[5] Thomas Ball and Sriram Rajamani. The slam toolkit. In Grard Berry, Hubert Comon, and Alain Finkel, editors, *Computer Aided Verification*, volume 2102 of *Lecture Notes in Computer Science*, pages 260–264. Springer Berlin / Heidelberg, 2001. 10.1007/3-540-44585-4-25.

[6] Chitta Baral and Yan Zhang. Knowledge updates: semantics and complexity issues. *Artificial Intelligence*, 164(1-2):209–243, 2005.

[7] Mordechai Ben-Ari, Zohar Manna, and Amir Pnueli. The temporal logic of branching time. In *Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '81, pages 164–176, New York, NY, USA, 1981. ACM.

[8] Patrick Blackburn, Maarten de Rijke, and Yde Venema. *Modal Logic*, volume 53 of *Cambridge Tracts in Theoretical Computer Scie*. Cambridge University Press, Cambridge, 2001.

[9] Marc Boyer and Mihaela Sighireanu. Synthesis and verification of constraints in the PGM protocol. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli,

editors, *FME 2003: Formal Methods*, LNCS 2805, pages 264–281. Springer-Verlag, 2003. financement europeen contract ADVANCE.

[10] F. Buccafurri, T. Eiter, G. Gottlob, and N. Leone. Enhancing model checking in verification by AI techniques. *Artificial Intelligence*, 112(1-2):57–104, 1999.

[11] F. Buccafurri, T. Eiter, G. Gottlob, and N. Leone. On ACTL formulas having linear counterexamples. *Journal of Computer and System Sciences*, 62(3):463–515, 2001.

[12] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 1020 states and beyond. *Inf. Comput.*, 98(2):142–170, 1992.

[13] Laura F. Cacovean, Emil M. Popa, and Cristina I. Brumar. Implementation of ctl model checker update. In *ICCOMP'07: Proceedings of the 11th WSEAS International Conference on Computers*, pages 432–437, Stevens Point, Wisconsin, USA, 2007. World Scientific and Engineering Academy and Society (WSEAS).

[14] Laura Florentina Cacovean and Florin Stoica. Ctl model update implementation using antlr tools. In *Proceedings of the WSEAES 13th international conference on Computers*, pages 169–174, Stevens Point, Wisconsin, USA, 2009. World Scientific and Engineering Academy and Society (WSEAS).

[15] Roberto Cavada, Alessandro Cimatti, Emanuele Olivetti, Marco Pistore, and Marco Roveri. *NuSMV 2.1 User Manual*, 2002.

[16] Roberto Cavada, Alessandro Cimatti, Emanuele Olivetti, Marco Pistore, and Marco Roveri. Nusmv, the symbolic model verifier, http://nusmv.fbk.eu/, 2011.

[17] Pankaj Chauhan, Edmund M. Clarke, James H. Kukula, Samir Sapra, Helmut Veith, and Dong Wang. Automated abstraction refinement for model checking large state spaces using sat based conflict analysis. In *FMCAD '02: Proceedings of the 4th International Conference on Formal Methods in Computer-Aided Design*, pages 33–51, London, UK, 2002. Springer-Verlag.

[18] Pi-Yu Chung, Yi-Min Wang, and Ibrahim N. Hajj. Logic design error diagnosis and correction. *IEEE Trans. VLSI Syst.*, 2(3):320–332, 1994.

[19] Alessandro Cimatti, Edmund M. Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. Nusmv 2: An opensource tool for symbolic model checking. In *CAV '02: Proceedings of the 14th International Conference on Computer Aided Verification*, pages 359–364, London, UK, 2002. Springer-Verlag.

[20] Alessandro Cimatti, Edmund M. Clarke, Fausto Giunchiglia, and Marco Roveri. NUSMV: A new symbolic model verifier. In *Proceedings of the 11th International Conference on Computer Aided Verification*, volume 1633, pages 495–499, 1999.

[21] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8:244–263, April 1986.

[22] Edmund Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded model checking using satisfiability solving. *Form. Methods Syst. Des.*, 19:7–34, July 2001.

[23] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003.

[24] Edmund Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, London, England, 1999.

[25] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, 1982. Springer-Verlag.

[26] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994.

[27] Edmund M. Clarke, Somesh Jha, Yuan Lu, and Helmut Veith. Tree-like counterexamples in model checking. In *Proceedings of the 17th Annual IEEE Sym-*

posium on Logic in Computer Science (LICS '02), pages 19–29, Washington, DC, USA, 2002. IEEE Computer Society.

[28] Edmund M. Clarke and Helmut Veith. Counterexamples revisited: Principles, algorithms, applications. In Nachum Dershowitz, editor, *Verification: Theory and Practice*, volume 2772 of *Lecture Notes in Computer Science*, pages 208–224. Springer, 2003.

[29] Edmund M. Clarke and Jeannette M. Wing. Formal methods: state of the art and future directions. *ACM Comput. Surv.*, 28:626–643, December 1996.

[30] Luca Console, Gerhard Friedrich, and Daniele Theseider Dupré. Model-based diagnosis meets error diagnosis in logic programs (extended abstract). In *Proceedings of the First International Workshop on Automated and Algorithmic Debugging*, pages 85–87, London, UK, 1993. Springer-Verlag.

[31] C. Daws and S. Yovine. Two examples of verification of multirate timed automata with kronos. In *In Proc. 1995 IEEE Real-Time Systems Symposium, RTSS'95*, pages 66–75. IEEE Computer Society Press, 1995.

[32] J de Kleer and B C Williams. Diagnosing multiple faults. *Artif. Intell.*, 32(1):97–130, 1987.

[33] Johan de Kleer. Local methods for localization of faults in electronic circuits. Cambridge: MIT Artificial Intelligence Laboratory Memo 394, 1976.

[34] Maria de Menezes, Silvio do Lago Pereira, and Leliane de Barros. System design modification with actions. In Antnio da Rocha Costa, Rosa Vicari, and Flavio Tonidandel, editors, *Advances in Artificial Intelligence SBIA 2010*, volume 6404 of *Lecture Notes in Computer Science*, pages 31–40. Springer Berlin / Heidelberg, 2011. 10.1007/978-3-642-16138-44.

[35] María del Mar Gallardo, Jesús Martínez, Pedro Merino, and Estefanía Rosales. Using xml to implement abstraction for model checking. In *SAC '02: Proceedings of the 2002 ACM symposium on Applied computing*, pages 1021–1025, New York, NY, USA, 2002. ACM.

[36] Stéphane Demri, François Laroussinie, and Ph. Schnoebelen. A parametric analysis of the state explosion problem in model checking. In *Proceedings*

of the 19th Annual Symposium on Theoretical Aspects of Computer Science, STACS '02, pages 620–631, London, UK, UK, 2002. Springer-Verlag.

[37] Louise A. Dennis, Raul Monroy, and Pablo Nogueira. Proof-directed debugging and repair. In Henrik Nilsson, editor, *Proceedings of the 7th Symposium on Trends in Functional Programming (TFP'06)*, pages 131–140, Nottingham, UK, April 19-21 2006.

[38] David L. Dill, Andreas J. Drexler, Alan J. Hu, and C. Han Yang. Protocol verification as a hardware design aid. In *Proceedings of the 1991 IEEE International Conference on Computer Design on VLSI in Computer & Processors*, ICCD '92, pages 522–525, Washington, DC, USA, 1992. IEEE Computer Society.

[39] Yulin Ding and David Hemer. An optimised algorithm to tackle the model explosion problem in ctl model update. In Byoung-Tak Zhang and Mehmet Orgun, editors, *PRICAI 2010: Trends in Artificial Intelligence*, volume 6230 of *Lecture Notes in Computer Science*, pages 589–594. Springer Berlin / Heidelberg, 2010. 10.1007/978-3-642-15246-754.

[40] Yulin Ding and Yan Zhang. Algorithms for ctl system modification. In Rajiv Khosla, Robert J. Howlett, and Lakhmi C. Jain, editors, *Knowledge-Based Intelligent Information and Engineering Systems*, volume 3682 of *Lecture Notes in Computer Science*, pages 1000–1006. Springer Berlin / Heidelberg, 2005. 10.1007/11552451138.

[41] Yulin Ding and Yan Zhang. A case study for ctl model update. In Jrme Lang, Fangzhen Lin, and Ju Wang, editors, *Knowledge Science, Engineering and Management*, volume 4092 of *Lecture Notes in Computer Science*, pages 88–101. Springer Berlin / Heidelberg, 2006. 10.1007/118112209.

[42] Yulin Ding and Yan Zhang. Ctl model update: Semantics, computations and implementation. In Gerhard Brewka, Silvia Coradeschi, Anna Perini, and Paolo Traverso, editors, *ECAI*, pages 362–366. IOS Press, 2006.

[43] Matthew B. Dwyer, Lori A. Clarke, Jamieson M. Cobleigh, and Gleb Naumovich. Flow analysis for verifying properties of concurrent software systems. *ACM Trans. Softw. Eng. Methodol.*, 13(4):359–430, 2004.

[44] E.M. Clarke, O. Grumberg, K.L. McMillan, and X. Zhao. Efficient Generation of Counterexamples and Witnesses in Symbolic Model Checking. In *32nd Design Automation Conference (DAC 95)*, pages 427–432, San Francisco, CA, USA, 1995.

[45] E. Allen Emerson and Edmund M. Clarke. Characterizing correctness properties of parallel programs using fixpoints. In *Proceedings of the 7th Colloquium on Automata, Languages and Programming*, pages 169–181, London, UK, 1980. Springer-Verlag.

[46] B. Berard et al. *Systems and software verification: model-checking techniques and tools*. Springer-Verlag New York, Inc., New York, NY, USA, 1999.

[47] Alexander Felfernig, Gerhard E. Friedrich, Dietmar Jannach, and Markus Stumptner. Exploiting structural abstractions for consistency based diagnosis of large configurator knowledge bases. In *Proceedings of the 14th European Conference on Artificial Intelligence (ECAI'2000) - Configuration Workshop*, pages 23–28, Berlin, Germany, 2000.

[48] Wan Fokkink, Jan Friso Groote, Jun Pang, Bahareh Badban, and Jaco van de Pol. Verifying a sliding window protocol in mucrl, 2003.

[49] B. A. Forouzan. *Data Communications and Networking*. McGraw-Hill Higher Education, New York, NY, USA, 3^{rd} edition, 2003.

[50] Python Software Foundation. The python programming language, http://www.python.org/, 2011.

[51] The Eclipse Foundation. The eclipse platform, http://www.eclipse.org/, 2011.

[52] Gordon Fraser and Franz Wotawa. Nondeterministic testing with linear model-checker counterexamples. In *QSIC '07: Proceedings of the Seventh International Conference on Quality Software*, pages 107–116, Washington, DC, USA, 2007. IEEE Computer Society.

[53] Gerhard Friedrich, Georg Gottlob, and Wolfgang Nejdl. Formalizing the repair process. In *ECAI '92: Proceedings of the 10th European conference on Artificial intelligence*, pages 709–713, New York, NY, USA, 1992. John Wiley & Sons, Inc.

[54] Gerhard Friedrich, Markus Stumptner, and Franz Wotawa. Model-based diagnosis of hardware designs. *Artificial Intelligence*, 111:3–39, 1996.

[55] Nikos Gorogiannis and Mark Ryan. Minimal refinements of specifications in modal and temporal logics. *Form. Asp. Comput.*, 19:417–444, October 2007.

[56] Andreas Griesmayer, Roderick Bloem, and Byron Cook. Repair of boolean programs with an application to c. In Thomas Ball and Robert B. Jones, editors, *CAV*, volume 4144 of *Lecture Notes in Computer Science*, pages 358–371. Springer, 2006.

[57] Alex Groce and Willem Visser. What went wrong: explaining counterexamples. In *Proceedings of the 10th international conference on Model checking software*, SPIN'03, pages 121–136, Berlin, Heidelberg, 2003. Springer-Verlag.

[58] Paulo Guerra and Renata Wassermann. Revision of ctl models. In Angel Kuri-Morales and Guillermo Simari, editors, *Advances in Artificial Intelligence IBERAMIA 2010*, volume 6433 of *Lecture Notes in Computer Science*, pages 153–162. Springer Berlin / Heidelberg, 2010. 10.1007/978-3-642-16952-616.

[59] Hannah Harris and Mark Ryan. Feature integration as an operation of theory change. In Frank van Harmelen, editor, *ECAI*, pages 546–550. IOS Press, 2002.

[60] Hannah Harris and Mark Ryan. Theoretical foundations of updating systems. *18th IEEE International Conference on Automated Software Engineering (ASE'03)*, 18th:291, 2003.

[61] Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-toi. Hytech: A model checker for hybrid systems. *Software Tools for Technology Transfer*, 1:460–463, 1997.

[62] Gerard Holzmann. *Spin model checker, the: primer and reference manual*. Addison-Wesley Professional, first edition, 2003.

[63] Gerard J. Holzmann. *Design and validation of computer protocols*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1991.

[64] Gerard J. Holzmann. The model checker SPIN. *Software Engineering*, 23(5):279–295, 1997.

[65] Shi-Yu Huang, Kwang-Ting Cheng, Kuang-Chien Chen, and Juin-Yeu Joseph Lu. Fault-simulation based design error diagnosis for sequential circuits. In *Proceedings of the 35th annual Design Automation Conference*, DAC '98, pages 632–637, New York, NY, USA, 1998. ACM.

[66] M. R. A. Huth and M. D. Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, Cambridge, England, 2000.

[67] Kurt Jensen. *Coloured Petri nets: basic concepts, analysis methods and practical use, vol. 2*. Springer-Verlag, London, UK, 1995.

[68] Barbara Jobstmann, Andreas Griesmayer, and Roderick Bloem. Program repair as a game. In Kousha Etessami and Sriram Rajamani, editors, *Computer Aided Verification*, volume 3576 of *Lecture Notes in Computer Science*, pages 287–294. Springer Berlin / Heidelberg, 2005. 10.1007/1151398823.

[69] Hyeong ju Kang and In cheol Park. Sat-based unbounded symbolic model checking. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, volume 24, 2005.

[70] Hirofumi Katsuno and Alberto O. Mendelzon. On the difference between updating a knowledge base and revising it. In *KR*, pages 387–394, 1991.

[71] M. Kelly, F. Pu, Y. Zhang, and Y. Zhou. Actl local model update with constraints. In *Knowledge-Based and Intelligent Information and Engineering Systems*, volume 6279, pages 135–144, 2010.

[72] Andreas Kuehlmann, David I. Cheng, Arvind Srinivasan, and David P. LaPotin. Error diagnosis for transistor-level verification. In *Proceedings of the 31st annual Design Automation Conference*, DAC '94, pages 218–224, New York, NY, USA, 1994. ACM.

[73] Kim G. Larsen, Paul Pettersson, and Wang Yi. Uppaal in a nutshell. *International Journal on Software Tools for Technology Transfer (STTT)*, 1:134–152, 1997. 10.1007/s100090050010.

[74] Dan Lawesson, Ulf Nilsson, and Inger Klein. Fault isolation using automatic abstraction to avoid state space explosion. In *Proceedings of the 18th Interna-*

tional Joint Conference - Artificial Intelligence Workshop on Model Checking and AI, pages 29–35, Acapulco, Mexico, 2003.

[75] K. McMillan. Symbolic model checking: an approach to the state explosion problem, 1992.

[76] Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic, 1993.

[77] Robert Meolic, Alessandro Fantechi, and Stefania Gnesi. Witness and counterexample automata for actl. In Manuel Núñez, Zakaria Maamar, Fernando L. Pelayo, Key Pousttchi, and Fernando Rubio, editors, *FORTE*, volume 3236 of *Lecture Notes in Computer Science*, pages 259–275. Springer, 2004.

[78] Debashis Nayak and D. M. H. Walker. Simulation-based design error diagnosis and correction in combinational digital circuits. In *Proceedings of the 1999 17TH IEEE VLSI Test Symposium*, VTS '99, pages 70–, Washington, DC, USA, 1999. IEEE Computer Society.

[79] Susan Owicki and Leslie Lamport. Proving liveness properties of concurrent programs. *ACM Trans. Program. Lang. Syst.*, 4:455–495, July 1982.

[80] Silvio Pereira and Leliane de Barros. A logic-based agent that plans for extended reachability goals. *Autonomous Agents and Multi-Agent Systems*, 16:327–344, 2008. 10.1007/s10458-008-9034-0.

[81] Malte Plath and Mark Ryan. Feature integration using a feature construct. *Sci. Comput. Program.*, 41(1):53–84, 2001.

[82] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in cesar. In *Proceedings of the 5th Colloquium on International Symposium on Programming*, pages 337–351, London, UK, 1982. Springer-Verlag.

[83] Robert L. Sedlmeyer, William B. Thompson, and Paul E. Johnson. Knowledge-based fault localization in debugging. *Journal of Systems and Software*, 3(4):301 – 307, 1983.

[84] Ehud Y. Shapiro. *Algorithmic Program Debugging*. MIT Press, Cambridge, MA, USA, 1983.

[85] Shengyu Shen, Ying Qin, and Sikun Li. Counterexample minimization for actl. In *CHARME'05*, pages 393–397. Springer-Verlag, 2005.

[86] Sharon Shoham and Orna Grumberg. A game-based framework for ctl counterexamples and 3-valued abstraction-refinement. *ACM Trans. Comput. Logic*, 9(1):1, 2007.

[87] S. Staber, B. Jobstmann, and R. Bloem. Finding and fixing faults. In *CHARME*, pages 35–49, 2005.

[88] P. Struss. Fundamentals of model-based diagnosis of dynamic systems. *IJCAI-97*, pages 480–485, 1997.

[89] M. Stumptner and F. Wotawa. A model-based approach to software debugging, 1996.

[90] Markus Stumptner and Franz Wotawa. Model-based program debugging and repair. In *In Proceedings IEA/AIE*, pages 155–160, 1996.

[91] Markus Stumptner and Franz Wotawa. A model-based tool for finding faults in hardware designs. In *In Proceedings Artificial Intelligence in Design*, 1996.

[92] Markus Stumptner and Franz Wotawa. A survey of intelligent debugging. *AI Commun.*, 11:35–51, January 1998.

[93] Raimund Ubar. Design error diagnosis with re-synthesis in combinational circuits. *Journal of Electronic Testing*, 19:73–82, 2003. 10.1023/A:1021948013402.

[94] Lionel van den Berg, Paul Strooper, and Wendy Johnston. An automated approach for the interpretation of counter-examples. *Electron. Notes Theor. Comput. Sci.*, 174:19–35, May 2007.

[95] Andreas Veneris and Jiang Brandon Liu. Incremental design debugging in a logic synthesis environment. *Journal of Electronic Testing*, 21:485–494, 2005. 10.1007/s10836-005-0335-9.

[96] Chao Wang, Gary D. Hachtel, and Fabio Somenzi. *Abstraction Refinement for Large Scale Model Checking (Series on Integrated Circuits and Systems)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[97] Chao Wang, Bing Li, HoonSang Jin, Gary D. Hachtel, and Fabio Somenzi. Improving ariadne's bundle by following multiple threads in abstraction refinement. In *ICCAD '03: Proceedings of the 2003 IEEE/ACM international conference on Computer-aided design*, page 408, Washington, DC, USA, 2003. IEEE Computer Society.

[98] Jeannette M. Wing and Mandana Vaziri-farahani. Model checking software systems: A case study. In *proceedings of the 3rd ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 128–139, 1995.

[99] Marianne Winslett. *Updating logical databases*. Cambridge University Press, New York, NY, USA, 1990.

[100] Bwolen Yang, Armin Biere, Sergio Campos, and Klaus Havelund. The nusmv example repository, http://nusmv.fbk.eu/examples/examples.html, 1999.

[101] Y. Zhang and Y. Ding. Ctl model update for system modifications. *Journal of Artificial Intelligence Research*, 31:113–155, 2008.

[102] Y. Zhang and M. Kelly. Local model update with an application to sliding window protocol. In *Knowledge-Based and Intelligent Information and Engineering Systems*, volume 6279, pages 11–21, 2010.

[103] Y. Zhang, M. Kelly, and Y. Zhou. Foundations of tree-like local model updates. In *ECAI*, volume 215, pages 615–620, 2010.

[104] Yefei Zhao, Zong yuan Yang, Jinkui Xie, and Qiang Liu. Formal model and analysis of sliding window protocol based on nusmv. *Journal of Computers*, 4(6), 2009.