A Case Study for CTL Model Update

Yulin Ding and Yan Zhang

School of Computing & Information Technology
University of Western Sydney
Kingswood, N.S.W. 1797, Australia
email: {yding,yan}@cit.uws.edu.au

Abstract. Computational Tree Logic (CTL) model update is a new system modification method for software verification. In this paper, a case study is described to show how a prototype model updater is implemented based on the authors' previous work of model update theoretical results [4]. The prototype is coded in Linux C and contains model checking, model update and parsing functions. The prototype is applied to the well known microwave oven example. This case study also illustrates some key features of our CTL model update approach such as the five primitive CTL model update operations and the associated minimal change semantics. This case study can be viewed as the first step towards the integration of model checking and model update for practical system modifications.

1 Introduction

As one of the most promising formal methods, automated verification has played an important role in computer science development. Currently, model checkers with SMV [2] or Promela [8] series as their specification languages are widely available for research, experiment, such as paper [11] and partial industry usage. Nowadays SMV, NuSMV [3], Cadence SMV [9] and SPIN [8] are well accepted as the state of the art model checkers. More recently, the MCK [5] model checker has added a knowledge operator to currently in use model checkers to verify knowledge related properties.

Buccafurri and his colleagues [1] applied AI techniques to model checking and error repairing. Harris and Ryan [6] proposed an attempt of system modification with a belief updating operator. Ding and Zhang [4] recently developed a formal approach called CTL model update for system modification, which was the first step towards a theoretical integration of CTL model checking and knowledge update. In this paper, we illustrate a case study of the microwave oven model to show how our CTL model updater can be used in practice to update the microwave oven example.

2 The Relationship between Model Checking and Model Update

Model checking is to verify whether a model satisfies certain required properties. Model checking is performed by the model checker. The SMV model checker was

first developed by McMillan [10] based on previous developed model checking theoretical results. This SMV model checker uses SMV as its specification language. Models and specification properties are all in the form of SMV language as the input. The SMV model checker parses the input into a structured representation for processing. Then, the system conducts model checking by SAT [2, 7] algorithms. The output is counterexamples which report error messages as the result of model checking. During the model checking, there was a state explosion problem, which significantly increases the SMV model checking search space. The introduction of OBDD [2, 7] in the SMV model updater solves the state explosion problem. After the first successful SMV compiler, the enhanced model checking compilers, NuSMV and Cadence SMV, were developed. NuSMV is an enhanced model checker from SMV and is more robust by the integration of a CUDD package [3]. It also supports LTL model checking. Cadence SMV was implemented for industrial use. The counterexample free concept is introduced in Cadence SMV. From SMV, NuSMV to Cadence SMV, the model checkers are developed from experimental versions to industrialized usage versions.

Model update is to repair errors in a model if the model does not satisfy certain properties. It is performed by the model updater. Our model updater updates the model after checking by the model checker if it does not satisfy the specification properties. The eventual output should be an updated model which satisfies the specification properties. In Fig. 1, the part of flow before "the original model" shows the model checking process. The part of flow after "The Original model" shows the model updater. The whole figure shows the complete process of model checking and model update.

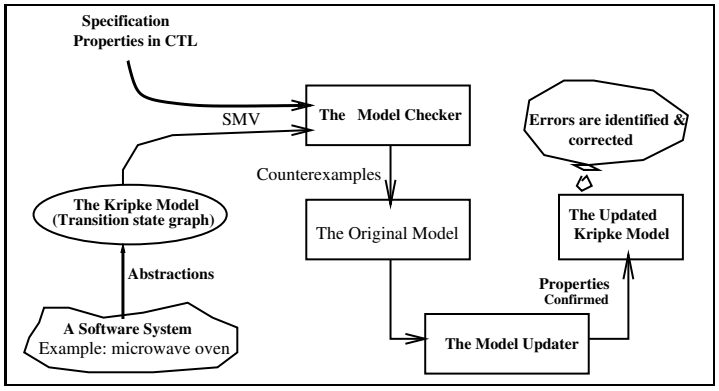

Fig. 1. The Model Checking and Model Update System

3 The Theoretical Principles of the CTL Model Updater

Ding and Zhang [4] have developed the theoretical principle of the model updater. The prototype of the model updater described later is implemented based on these results. Before we introduce the CTL model updater, we review the CTL syntax and semantics and the theoretical results of CTL model update.

3.1 CTL Syntax and Semantics

Definition 1. *[2] Let AP be a set of atomic propositions. A Kripke model M over AP is a three tuple $M = (S, R, L)$ where 1. S is a finite set of states. 2. $R \subseteq S \times S$ is a transition relation. 3. $L : S \to 2^{AP}$ is a function that assigns each state with a set of atomic propositions (named variables in our system).*

Definition 2. *[7] Computation tree logic (CTL) has the following syntax given in Backus naur form (only listed syntax related to the case study in this paper):*

$\phi ::= p|(\neg\phi)|(\phi \wedge \phi)|(\phi \vee \phi)|AG\phi|EG\phi|AF\phi|EF\phi$

where p is any propositional atom.

Definition 3. *[7] Let $M = (S, R, L)$ be a Kripke model for CTL. Given any s in S, we define whether a CTL formula ϕ holds in state s. We denote this by $M, s \models \phi$. Naturally, the definition of the satisfaction relation $\models$ is done by structural induction on all CTL formulas (only listed semantics related to the case study in this paper):*

1. *$M, s \models p$ iff $p \in L(s)$.*
2. *$M, s \models \neg\phi$ iff $M, s \not\models \phi$.*
3. *$M, s \models \phi_1 \wedge \phi_2$ iff $M, s \models \phi_1$ and $M, s \models \phi_2$.*
4. *$M, s \models \phi_1 \vee \phi_2$ iff $M, s \models \phi_1$ and $M, s \models \phi_2$.*
5. *$M, s \models AG\phi$ holds iff for all paths $s_0 \to s_1 \to s_2 \to \cdots$, where s_0 equals s, and all s_i along the path, we have $M, s_i \models \phi$.*
6. *$M, s \models EG\phi$ holds iff there is a path $s_0 \to s_1 \to s_2 \to \cdots$, where s_0 equals s, and for all s_i along the path, we have $M, s_i \models \phi$.*
7. *$M, s \models AF\phi$ holds iff for all paths $s_0 \to s_1 \to s_2 \to \cdots$, where s_0 equals s, there is some s_i such that $M, s_i \models \phi$.*
8. *$M, s \models EF\phi$ holds iff there is a path $s_0 \to s_1 \to s_2 \to \cdots$, where $s_i = s$,and for some s_i along the path, we have $M, s_i \models \phi$.*

3.2 CTL Model Update with Minimal Change

Definition 4. *[4] (**CTL Model Update**) Given a CTL Kripke model $M = (S, R, L)$ and a CTL formula ϕ such that $\mathcal{M} = (M, s_0) \not\models \phi$, where $s_0 \in S$. An* update *of $\mathcal{M}$ with ϕ, is a new CTL Kripke model $M' = (S', R', L')$ such that $\mathcal{M}' = (M', s'_0) \models \phi$ where $s'_0 \in S'$. We use $Update(\mathcal{M}, \phi)$ to denote the result $\mathcal{M}'$.*

The operations to update the CTL model can be decomposed into 5 atomic updates called primitive operations in [4]. They are the foundation of our prototype for model update and are denoted as PU1, PU2, PU3, PU4 and PU5. PU1: adding a relation only; PU2: removing a relation only; PU3: substituting a state and its associated relation(s) only; PU4: adding a state and its associated relation(s) only; PU5: removing a state and its associated relation(s) only. Their mathematical specifications are in [4]. Model update should obey minimal change rules, which are described as follows.

Given models $M = (S, R, L)$ and $M' = (S', R', L')$, where M' is an updated model from M by only applying operation PUi on M. we define $Diff_{PUi}(M, M') = (R - R') \cup (R' - R)$ $(i = 1, 2)$, $Diff_{PUi}(M, M') = (S - S') \cup (S' - S)$ $(i = 3, 4, 5)$ and $Diff(M, M') = (Diff_{PU1}(M, M'), \cdots, Diff_{PU5}(M, M'))$.

Definition 5. *[4](***Closeness Ordering***) Given three CTL Kripke models M, M_1 and M_2, where M_1 and M_2 are obtained from M by applying $PU1 - PU5$ operations. We say that M_1 is closer or as close to M as M_2. denoted as $M_1 \leq_M M_2$, iff $Diff(M, M_1) \preceq Diff(M, M_2)$. We denote $M_1 <_M M_2$ if $M_1 \leq_M M_2$ and $M_2 \not\leq_M M_1$.*

Definition 6. *[4] (***Admissible Update***) Given a CTL Kripke model $M = (S, R, L)$, $\mathcal{M} = (M, s_0)$ where $s_0 \in S$, and a CTL formula ϕ, $Update(\mathcal{M}, \phi)$ is called* admissible *if the following conditions hold: (1) $Update(\mathcal{M}, \phi) = (M', s_0') \models \phi$ where $M' = (S', R', L')$ and $s_0' \in S'$; and (2) there does not exist another resulting model M'' such that $(M'', s_0'') \models \phi$ and $M'' <_M M'$.*

4 The Prototype of the CTL Model Updater

We have simulated a prototype of the CTL model updater in Linux C as the implementation of our algorithms. Unlike SMV, the input models are pre-specified in C code. Our system does not contain OBDD [7] optimization as the SMV mode updater. Thus, there is not excessive processing load for our prototype as with the SMV compiler for its parsing and checking phases. We have coded our own model checking functions to perform the model checking duty during the update process. The CTL model updater includes library functions, predefined model definition functions, a specification string parser, model checking functions and model update functions. The diagram of the code structure is shown in Fig 2. A detailed description of the system follows.

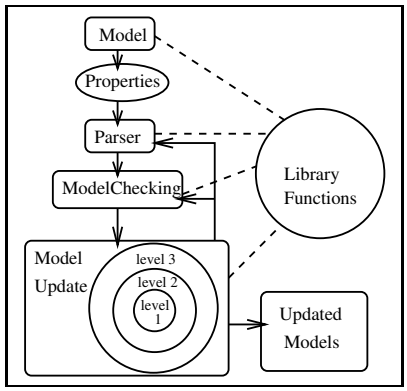

Fig. 2. The flow diagram of the Model Update System

4.1 Predefined Structures and Library Functions

We have coded a set of pre-defined structures for the whole system. The most significant structures are the model definition structure, the state structure, the state data structure, and the atom and calc_pair structures for storing specification string parsing results.

The model definition structure contains the major elements of a CTL model. The definition structure contains a state pointer array and a state count, where each reachable state is defined in a state structure. The structure contains the names and number of the defined variables. The structure contains a path pointer array and a path count, where each path is defined in a path structure. The path is a structure containing a state count and array of state pointers. The structure in C code is as Fig 3. In this structure, "name" is the name of a model; "numvar" is the number of variables; "varname" is an array of variable names in a model; "numstates" is the number of non repeated reachable states in a model. "state[MAXSTATE]" is an array of pointers to state structures containing each non repeated reachable state; "numpaths" is the number of paths in a model; "path[MAXPATH]" is an array of pointers to the defining path structures. In our CTL model updater, the model definition structure is defined as an static instance. The change due to update on a model is eventually stored in the definition instance.

```
typedef struct {
        char name[MAXCHAR];
        int numvar;
        char
varname[MAXVAR][MAXCHAR];
        int numstates;
        state_ptr state[MAXSTATE];
        int numpaths;
        path_ptr path[MAXPATH];
} state_defn;
```

Fig. 3. The state definition structure

```
typedef struct {
        int num;
        boolean initial;
        boolean var[MAXVAR];
        int numnext;
        int next[MAXTRANS];
        int numprev;
        int prev[MAXTRANS];
        boolean result;
} state;
```

Fig. 4. The state structure in C code

The state structure is the major component defining a model. The state structure contains all information in a state in particular the values of the variables of the state, and the relations in between this state and its previous or successive states. The state structure is defined as Fig. 4. In this structure, "num" is an identifier as an integer of a state; "initial" is a boolean variable to define this state as an initial state in the model; "var[MAXVAR]" is an array of boolean variable values for this state; "numnext" is the total number of next states; "next[MAXTRANS]" is an array of the integer identifiers of next states; "numprev" is the total number of previous states; prev[MAXTRANS] is an array of the integer indentifiers of previous states. "result" is a boolean variable to store the checking result for the state.

Another major structure called "state_data" is an interface structure to actually load a state structure.

The library functions include all initializations of the model in the definition structure, simple operations for model checking and update, and printing functions for a model and its paths. For the initializations, there are functions for defining a model, its name and states, setting data in states, setting and clearing links in between states and so on. The simple operations for model checking and update include checking individual and all states in a model, checking a path or all paths in a model, adding or removing states, building or removing links in between states and calculating paths etc. The printing functions include printing states, paths and the model. The printing functions assist the user in understanding the operations performed by the model updater.

4.2 Parser

The parsing functions decompose a complex CTL formula, expressed as a string, into a number of linked structures. The components of the structures have direct equivalence to each recognizable component of the specification string as our case study illustrates below. For our system the part which needs to be parsed is the string representing the specification property, such as the property in the microwave oven model: "¬EF(Start∧ EG¬Heat)". Our parser rationalizes a CTL specification string according to the Backus Naur form [7] expressed as definition 2. There are two major structures used by our parsing library functions which store our parsing results.

An atom structure (Fig. 5) stores the results of parsing a symbol ϕ expression including ¬ and path navigation expressions. An atom structure assumes that the string contains semantics such as AG, EG and so on with a boolean atomic variable successor. In Fig. 5, "negate1" is the negation symbol in front of "navigate" (such as AG or EG); if "negate1" is true, the negation symbol in front of "navigate" is there, otherwise, there is not a negation symbol; "negate2" is the negation symbol after "navigate". It behaves the same as "negate1"; "navigate"

```
typedef struct {
        boolean negate1;
        boolean negate2;
        int navigate;
        int varindex;
        boolean error;
} atom;
```

Fig. 5. The atom structure in C

```
typedef struct {
        boolean negate1;
        boolean negate2;
        int navigate;
        optype operator;
        atom_ptr operand1;
        atom_ptr operand2;
        void * nestedpair1;
        void * nestedpair2;
        boolean error;
} calc_pair;
```

Fig. 6. The pair structure in C

is the semantics about the model such as “AG” or “EG”. We define numbers to represent different semantics. For example, “AF” is 4, “AG” is 5 and “EG” is 6; “varindex” is the index number of the variables in our system and represents the index position of the variable in the model definition object but includes an adder to avoid conflict with other indexes, which serves for our code only; “error” indicates whether the atom parsed correctly or not. If “error” is true, it means that the atom may not exist in our model. For example, if a string is “Start”, which is a name of a variable in the model, then the structure of the parsed string should be the part of components after “operand1” and before “operand2” in Fig. 7. If a string is “EG¬ Heat”, where “Heat” is a name of the variables in a model, then the structure of the parsed string is the part of components after “operand2” and before “nestedpair1” in Fig. 7.

primary pair →
- negate1 ··· false
- negate2 ··· false
- navigate ··· 5
- operator ··· 0
- operand1 ··· 0x00000000
- operand2 ··· 0x00000000
- nested pair1 →
  - negate1 ··· true
  - negate2 ··· false
  - navigate ··· 0
  - operator ··· 22
  - operand1 →
    - negate1 ··· false
    - negate2 ··· false
    - navigate ··· 0
    - varindex ··· 101
    - error ··· false
  - operand2 →
    - negate1 ··· false
    - negate2 ··· true
    - navigate ··· 6
    - varindex ··· 103
    - error ··· false
  - nestedpair1 ··· 0x00000000
  - nestedpair2 ··· 0x00000000
  - error ··· false
- nestedpair2 ··· 0x00000000
- error ··· false

Fig. 7. The parsed structure for string “AG(¬(Start∧ EG¬Heat))”

primary pair →
- negate1 ··· false
- negate2 ··· false
- navigate ··· 0
- operator ··· 23
- operand1 →
  - negate1 ··· true
  - negate2 ··· false
  - navigate ··· 0
  - varindex ··· 101
  - error ··· false
- operand2 →
  - negate1 ··· true
  - negate2 ··· true
  - navigate ··· 6
  - varindex ··· 103
  - error ··· false
- nestedpair1 ··· 0x00000000
- nestedpair2 ··· 0x00000000
- error ··· false

Fig. 8. The parsed structure for string “$\neg Start \vee \neg EG \neg Heat$”

A pair structure stores results of parsing an expression containing two ϕ expressions and a separating operator. This structure includes storage for a path

navigation expression and leading and following negate declarations. If the structure of a string is more complex than an atom, then it needs to be expressed in a pair structure in Fig. 6. In this structure, "negate1", "negate2", "navigate" and "error" are the same concepts as those in the atom structure; "operator" is a logic symbol such as "$\wedge$" or "$\vee$" and is defined as an integer in the structure; "operand1" is the "atom" before "operator"; "operand2" is the "atom" after "operator"; "nestedpair1" ("nestedpair2") is a casted type of "calc_pair" if the string before (or after) "operator" is a "calc_pair", which can accommodate recursively nested "calc_pair" structures; For example, the string "AG($\neg$(Start$\wedge$ EG$\neg$Heat))" can be parsed into the "calc_pair" structure as in Fig. 7. In this figure, the elements before "nested pair1" match AG in the given string; the elements after "nested pair1" and before "operator" are the "$\neg$" after "AG"; "operator$\cdots$ 22" is the "$\wedge$" in between "Start" and "EG$\neg$ Heat"; the elements after "operand1" and "operand2" are atoms which have been explained before the pair structure description. During model checking and update, we select the needed elements for any parts of the string from the corresponding parsed structure.

The parser also contains a set of functions to rationalize negate symbols (normalize) in a specification to simplify processing. These functions use the parsing structures as input and output.

4.3 Model Checking Functions

The model checking functions are for checking CTL semantics, such as whether "AG", "EG", are true or not. They are continually used for the whole process of update. Before or after each step of update, they are called to do model checking and identify error or correct states according to different semantics or update requirements. Atomic model checking functions deal with model checking for atomic variables only. In our model checking functions, we have checking functions with "true" or "false" results to tell whether a specification property satisfies CTL semantics. To assess whether a state satisfies the required property or not, we compare the variables in a state with the variables in the required property. For particular semantics such as "EG", its model checking function is performed for each path, where each state is checked. If all states on at least one path satisfy the required property, then it means the model checking is "true" with semantic "EG". Besides, we also have functions which identify error or correct paths or states for particular CTL semantics, which will be used for model updating. For example, for semantics "EG", there are functions to identify correct or error states in a model or correct or error paths in a model. The information contained in these functions is the state or path identification numbers. If model update functions use them, they can locate the error paths or states straight away to perform model update on these states or related relations.

4.4 Model Update Functions

The model updating functions are the most important part of the system and demonstrate our previous theoretical results. They are called to update the model

either on paths (eventually on states of the path) or states among all reachable states. The update functions frequently call model checking functions for each step update to see whether the updated model satisfies certain features or not. If the updated model satisfies the required feature, then the update is halted and the system returns updated models. The update obeys our minimal change rules. The resulting model could be more than one if they are not interchangeable. If the update changes the model, the definition structure containing the model is changed as well.

The update functions include atomic updates (level 1) PU1 to PU5, which update single states and their relations, and atomic update for variables in a state: adding or removing (changing) a variable. Above the atomic updates, we have 2nd level update functions for updating the semantics of a model such as AG, EG etc.. Above the 2nd level update functions, we have the outer level (level 3) update functions which are the combination of parsing, model checking and updating if the input string is not an atomic variable.

If the string representing the required property is not an atomic variable, then we parse the string before doing model checking and update. For example, if the input required property is AG(Start$\wedge\neg$ Error), then all states in a model should satisfy the string after AG. The string, "Start$\wedge\neg$ Error", should be parsed before further update for each state on this model. This process is performed by the functions at our 2nd and 3rd level updates which call the 1st level functions. During the process, parsing string and nested model checking and update involves certain degrees of intelligent reasoning depending on the semantics and complexity of the string. The reasoning is done by update functions on the 3rd level. If a required property is in a form such as "AG(Start)" where Start is a variable, then it can be performed by the 2nd level update functions which eventually call the 1st level update functions.

5 The Microwave Oven Model

The microwave oven model has a total of $2^4 = 16$ states, where there are 7 reachable states and one initial state, and 4 variables with boolean values. The Kripkle model of the microwave oven [2] is in Fig. 9, which shows its 7 reachable states $\{s_1, s_2, s_3, s_4, s_5, s_6, s_7\}$ and their 12 relations. The set of variables is $\{Start, Close, Heat, Error\}$ and each variable has boolean values. The specification property is "$\neg EF(Start \wedge EG\neg Heat)$". The result of model checking shows that the model does not satisfy the specification property. Our model updater will update the model and the updated models will satisfy the specification property.

The model is stored in an instance of the model definition structure. The specification is predefined in a char array (string). First, we should parse the specification string "$\neg EF(Start\wedge EG\neg Heat)$" into a parsing structure. Then, we convert the structure into a new structure corresponding to specification formula $AG(\neg(Start \wedge EG\neg Heat))$ to remove the front $\neg$. The conversion is performed by a normalize function. The parsing structure of the string "$AG(\neg(Start \wedge EG\neg Heat))$" is shown in Fig. 7.

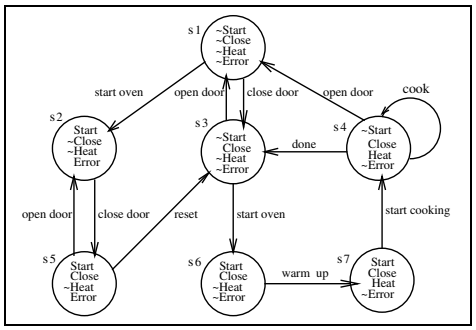

Fig. 9. The Original CTL Kripke Structure of a Microwave Oven

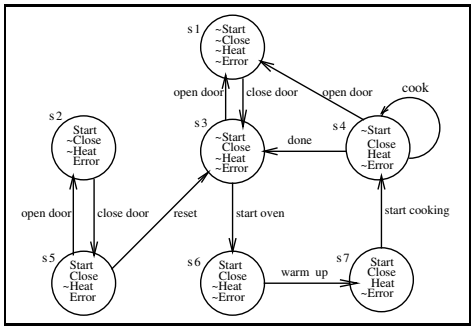

Fig. 10. The Updated Microwave Oven Model with Primitive Update PU2

Then, we must check each state's variables (because of AG) according to the property $\neg(Start \wedge EG\neg Heat)$ which is a nested calc_pair in our parsing structure. This is performed by a model checking process for AG which is called by level 3 update functions. We select $EG\neg Heat$ after $\wedge$" to update first, whose parsed elements are under "operand2" of "nested pair1" in Fig. 7, to apply model checking functions to identify a path (or paths) for which EG is valid. In this model, any path which has each state with variable $Heat$ false should be identified. Here, we find the paths $s_1 \rightarrow s_2 \rightarrow s_5 \rightarrow s_3 \rightarrow s_1 \cdots$ and $s_1 \rightarrow s_3 \rightarrow s_1 \cdots$ which are Strongly Connected Components (SCC) loops [2] satisfying $EG\neg Heat$. Then, we check where the states have variable $Start$ true, which is the atomic string before $\wedge$ in the specification string and maps the elements in between "operand1" and "operand2" under "nested pair1" in the parsed structure in Fig. 7. We identify states s_2, s_5,s_6 and s_7 with $Start$ true by model checking functions for AG because before $\wedge$ "$Start$" is atomic and the "AG" before "$Start$" should be mapped as the semantic symbol in front of "Start". Now, we must identify states which have both variables $Start$ true and $Heat$ false because of the "$\wedge$" operator between "Start" and "EG$\neg$Heat". These states are s_2 and s_5. It means that the two states satisfy $Start \wedge EG\neg Heat$. However, the $AG(\neg$ before them in $AG(\neg(Start \wedge EG\neg Heat))$ specifies that the model should not have any state which satisfies this feature. Thus, we must update s_2 and s_5.

Now, the 2nd level update function for AG calls atomic (1st level) update functions such as PU1-PU5. The results are three equal minimal updates: for the atomic update PU2 case, relation (s_1, s_2) is deleted; for the atomic update PU5 case, state s_2 and relations (s_1, s_2), (s_2, s_5) and (s_5, s_2) are deleted; for the PU3 case, we must normalize the part of string after "AG" before PU3 is performed. $\neg(Start \wedge EG\neg Heat) = \neg Start \vee \neg EG\neg Heat$. The corresponding parsed structure for $\neg Start \vee \neg EG\neg Heat$ is as Fig. 8:

Thus, eventually the faulty states s_2 and s_5 should be updated with either $\neg Start$ or $\neg EG\neg Heat$ in an update function for the $\vee$ operator. Obviously, $\neg Start$ is simpler thus is chosen. As we mentioned, the selection process involves certain intelligent reasoning.

After these updates, the resulting model $\mathcal{M}'= (M', s_1) \models \neg EF(Start \wedge EG\neg Heat)$. The above three resulting models are all minimally changed from the original model and are admissible. They are not interchangeable with each other due to our minimal change rules. The updated models are shown as Fig. 10, 11 and 12.

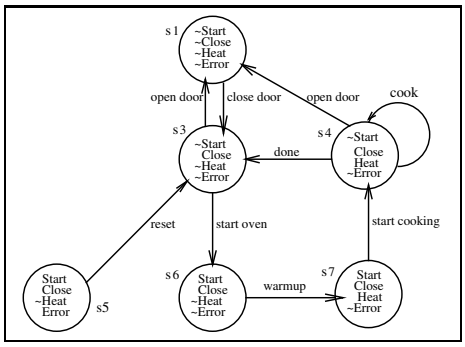

Fig. 11. The Updated Microwave Oven Model with Primitive Update PU5

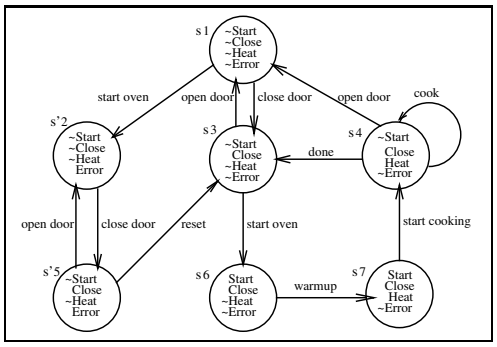

Fig. 12. The Updated Microwave Oven Model with Primitive Update PU3

6 The Simulation Results for Updating the Microwave Oven Model

We show partial screen results by running the executable file as follows. In the beginning, the screen shows the model name, variables, states and relations in between states:

```
State Machine Model: Model name is Microwave Oven

        Variable name #1 is Start
        Variable name #2 is Close
        Variable name #3 is Heat
        Variable name #4 is Error

State Information for 7 states is ->
Id Initial       Values          Next Links       Previous Links
 1   ***  false false false false  -> 2 -> 3       <- 4 <- 3
 2        true  false false true   -> 5            <- 1 <- 5
 3        false true  false false  -> 6 -> 1       <- 1 <- 5 <- 4
 4        false true  true  false  -> 3 -> 1 -> 4   <- 7
 5        true  true  false true   -> 2 -> 3       <- 2
 6        true  true  false false  -> 7            <- 3
 7        true  true  true  false  -> 4            <- 6
```

We omit parsed structure and paths here. The states which must be updated are identified as s_2 and s_5. We only demonstrate three admissible updated results as follows.

```
Case 1: after PU2 update on the relation between state 1 & 2

State Information for 7 states is ->
 Id Initial        Values           Next Links      Previous Links
  1   ***  false false false false  -> 3            <- 4 <- 3
  2        true  false false true   -> 5            <- 5
  3        false true  false false  -> 6 -> 1       <- 1 <- 5 <- 4
  4        false true  true  false  -> 3 -> 1 -> 4  <- 7
  5        true  true  false true   -> 2 -> 3       <- 2
  6        true  true  false false  -> 7            <- 3
  7        true  true  true  false  -> 4            <- 6
```

This output demonstrates the removal of the s_1 to s_2 state transition.

```
 Case 2: after PU5 update on states 2 & 5

 State Information for 6 states is ->
 Id Initial        Values           Next Links      Previous Links
  1   ***  false false false false  -> 3            <- 4 <- 3
  3        false true  false false  -> 6 -> 1       <- 1 <- 5 <- 4
  4        false true  true  false  -> 3 -> 1 -> 4  <- 7
  5        true  true  false true   -> 3
  6        true  true  false false  -> 7            <- 3
  7        true  true  true  false  -> 4            <- 6
```

This output demonstrates the removal of s_2 and its associated links.

```
 Case 3: after PU3 update on states 2 & 5

 State Information for 7 states is ->
 Id Initial        Values           Next Links      Previous Links
  1   ***  false false false false  -> 3 -> 22      <- 4 <- 3
 22        false false false true   -> 55           <- 1 <- 55
  3        false true  false false  -> 6 -> 1       <- 1 <- 4 <- 55
  4        false true  true  false  -> 3 -> 1 -> 4  <- 7
 55        false true  false true   -> 3 -> 22      <- 22
  6        true  true  false false  -> 7            <- 3
  7        true  true  true  false  -> 4            <- 6
```

This output demonstrates the modification of s_2 and s_5 (re-identified as 22 and 55) with updated variable values. 22 is $s'2$ and 55 is $s'5$ in Fig. 12.

7 Conclusions and Future Work

In this paper, we have demonstrated the implementation of model update theory and minimal change rules with a prototype based on the well known microwave

oven example. It is an important step to advance model update from theoretical research to practice. At this stage, after we have successfully demonstrated the microwave oven example, we are coding another two well known examples: afs0 and afs1 models [11]. We intend to apply our model updater to these models as well to demonstrate hosting a more complex model with a larger number of states.

We are targeting to a formal CTL model update compiler which can accept SMV as input. Thus, our intention is that counterexamples from the existing SMV model checker will be used as part of the input of the model updater. The internal integration of our model update philosophy and the existing SMV model checker requires a comprehensive coding effort. This effort is a major future milestones for system modification and will significantly improves the usage of the SMV model checker.

8 Acknowledgement

The authors thank senior software engineer Neville Cockburn for his important guidance and help for this system implementation.

References

1. Buccafurri, F., Eiter, T., Gottlob, G. and Leone, N. (1999). Enhancing model checking in verification by AI techniques. *Artificial Intelligence* 112(1999) 57-104.
2. Clarke, E. Jr. et al. (1999). *Model Checking*, The MIT press, Cambridge, Massachusetts, London, England. ISBN 0-262-03270-8, Pp. 314.
3. Cimatti, A. et al. (1999). NUSMV: a new symbolic model verifier. In *Proceedings of the 11th International Conference on Computer Aided Verification*. Vol. 1633 in LNCS. Pp.495-499.
4. Ding,Y. and Zhang,Y.(2005). Model Update CTL Systems. In proceedings of The 18th Australian Joint Conference on Artificial Intelligence. Sydney, December, 2005. Pp.1-12.
5. Gammie, P. and van der Meyden, R.(2004). MCK-Model checking the logic of knowledge. In *the Proceeding of the 16th International Conference on Computer Aided Verification*. Pp. 479 - 483.
6. Harris,H. and Ryan,M. (2003). Theoretical foundations of updating systems. In *the Prodeeding of the 18th IEEE International Conference on Automated Software Engineering*. Pp.291-298.
7. Huth, M. and Ryan, M. (2000). *Logic in Computer Science: Modelling and Reasoning about Systems*. University Press, Canbridge.
8. Holzmann, Gerard. (2003). The SPIN Model Checking: Primer and Reference Manual. Addison-Wesley Professional. ISBN: 0321228626. Pp.596.
9. McMillan,K. and Amla,N. (2002). Automatic abstraction without counterexamples. Cadence Berkeley Labs, Cadence Design Systems.
10. McMillan,K. (1992). The SMV System. http://www.cs.cmu.edu/ modelcheck/smv.html
11. Wing,J. and Vaziri-Farahani, M.(Oct.1995). A case study in model checking software. In proceedings of 3rd ACM SIGSOFT Symposium on the Foundations of Software Engineering.