A Combined Formal Analysis Methodology and Towards Its Application to Hierarachical State Transition Matrix Designs

Weiqiang Kong
Graduate School of IS&EE,
Kyushu University, Japan.
weiqiang@qito.kyushu-u.ac.jp

Leyuan Liu
Graduate School of IS&EE,
Kyushu University, Japan.
leyuan@f.ait.kyushu-u.ac.jp

Hirokazu Yatsu
Graduate School of IS&EE,
Kyushu University, Japan.
hirokazu.yatsu@f.ait.kyushu-u.ac.jp

Akira Fukuda
Graduate School of IS&EE,
Kyushu University, Japan.
fukuda@ait.kyushu-u.ac.jp

***Abstract*—Interactive theorem proving and model checking are known as two formal verification techniques that have complementary features and aims, but overlapping application areas. In this paper, we investigate a procedure (methodology) called Combined Falsification and Verification (CFV), by which the benefits of both interactive theorem proving and model checking could be enjoyed for formal analysis of software systems against invariant properties. We have been developing a SMT-based Bounded Model Checker called Garakabu2 for falsification of HSTM designs. Interfaces necessary for enabling the procedure CFV is planned to be introduced into Garakabu2 for providing an auxiliary functionality for users of Garakabu2 who are experts in formal methods.**

***Keywords*-Interactive theorem proving; Bounded Model Checking; Invariant Properties; State Transition Matrix.**

I. INTRODUCTION

As software systems grow in scale and functionality, there is an increasing demand that these systems should be reliable. This is especially the case for those systems that are safety-critical ones, such as banking systems, railway systems and aircraft guidance systems, in which subtle errors can cause fatal losses in economy and lives. One way of improving reliability of software systems is by using formal verification, which are mathematically-based techniques for specifying and verifying systems.

Interactive theorem proving [1] and model checking [2] are known as two formal-verification techniques that have complementary features and aims, but overlapping application areas. The main different characteristics between them lie with the aspects of state space (infinite vs. finite), automation (limited vs. fully), and counterexample (not automatic vs. automatic). There are no hard-and-fast answers to the priority of one to the other. It is a common consensus that the two techniques are equally effective and maintainable on the whole for complex applications, while each technique has specific strengths and weaknesses. By combining them, attempts have been made to enjoy the best of both worlds, but no comprehensive understanding exists.

In this paper, we pursue a better understanding about the combination of the two techniques through a specific combination that the OTS/CafeOBJ method [3] is backended with Maude model checkers [4]. Specifically, we focus on how the counterexamples automatically generated by model checkers can help the interactive inductive verification technique of theorem proving for invariant properties.

Particularly, regarding how to combine the two techniques, we have previously proposed a procedure called Induction-Guided Falsification (IGF) in [5]. IGF is a procedure that can reveal logical errors (i.e., falsification) lurking in the specifications for theorem proving as early as possible by employing model checking during the interactive inductive verification of invariant properties. As an extension of IGF with respect to verification, we investigate a procedure called Combined Falsification and Verification (CFV). CFV is a more general procedure that combines interactive inductive verification of invariant properties with model checking, which is supposed to be followed by human verifiers.

We have been developing a SMT-based [6] Bounded Model Checking (BMC) [7] tool called Garakabu2 [8], [9] for formal analysis of designs specified in Hierarchical State Transition Matrix (HSTM) [10], a set of State Transition Matrices organized in a hierarchical structure. HSTM has been widely accepted and used by particularly Japanese embedded software industry, and been adopted as the modeling language of commercial model-based tools. However, one issue is that Garakabu2 only conducts falsification due to that only a bounded state space is checked with BMC. We plan to implemented interfaces necessary for enabling the procedure CFV, and thus, make Garakabu2 usable for conducting formal analysis of HSTM designs with both interactive theorem proving and model checking.

The paper is organized as follows. Section II introduces preliminary knowledge. Section III first reviews IGF and then proposes the procedure CFV. Section IV investigates briefly the feasibility/possibility of introducing CFV into Garakabu2, and Section V concludes the paper.

II. PRELIMINARY

The procedures IGF and CFV are proposed and described based on the combination of the OTS/CafeOBJ method (for interactive theorem proving) and Maude Model Checkers. In this section, we informally review these two methods/techniques, while refer readers to [3] and [4] for

Copyright (c) IARIA, 2012. ISBN: 978-1-61208-233-2

their respective formal details. We use a simple mutual exclusion algorithm using a queue to demonstrate how to use the OTS/CafeOBJ method, and respectively, the Maude LTL model checker to specify and verify invariant properties. The pseudo-code executed by each process i repeatedly can be described as follows:

$l1$: $put(queue, i)$;
$l2$: **repeat until** $top(queue) = i$
critical section;
cs : $get(queue)$;

$queue$ is the queue of process IDs shared by all processes. $put(queue, i)$ puts a process ID i into *queue* at the end, $get(queue)$ deletes the top element from $queue$, and $top(queue)$ returns the top element of $queue$. Each iteration of the loop at label l2 is supposed to be atomically processed. Initially each process i is at label *l1* and $queue$ is *empty*.

A. The OTS/CafeOBJ Method

The OTS/CafeOBJ method [3] is a modeling, specification and verification method. In the OTS/CafeOBJ method, a system to be verified is first modeled as an observational transition system (OTS), a transition system that can be straightforwardly written in terms of equations. The OTS is then written in CafeOBJ [11] as a behavioral specification (Some basic data types used in the OTS, such as `Nat` and `Int` are described as general algebraic specifications, which are imported in this behavioral specification).

The OTS/CafeOBJ specification of the sample mutual exclusion algorithm consists of three data type modules (with the names `LABEL`, `PID` and `QUEUE`) and one OTS module (with the name `QLOCK`). The three data type modules define sorts `Label`, `Pid` and `Queue`, respectively. We show module `LABEL` as an example and the other two are defined similarly. `Label` is written in CafeOBJ as:

```
mod! LABEL {
  [Label]
  ops l1 l2 cs : -> Label
  op _=_ : Label Label -> Bool {comm}
  var L : Label
  eq (L = L) = true .      eq (l1 = l2) = false .
  eq (l1 = cs) = false .  eq (l2 = cs) = false .
}
```

In the module `LABEL`, `[Label]` is the declaration of the sort `Label`; `l1`, `l2` and `cs` are declared constants; `L` is a declared variable. Note that operator _=_ is the equality predicate for sort `Label`.

The OTS module specifies behaviors (state transitions) of the algorithm. The sort denoting states of the OTS is declared as `Sys`. The operators denoting the observers and transitions are declared as follows (where '`--`' marks the rest of the line as a comment):

```
-- observers
bop pc    : Sys Pid -> Label
bop queue : Sys     -> Queue
-- transitions
bop want  : Sys Pid -> Sys
bop try   : Sys Pid -> Sys
bop exit  : Sys Pid -> Sys
```

`Pid`, `Label` and `Queue` are the sorts denoting process IDs, labels and queues of process IDs, respectively. The corresponding data type modules (`LABEL`, `PID` and `QUEUE`) defining these three sorts are imported in the OTS module.

Let `I`, `J` be CafeOBJ variables for `Pid`, and `S` be a CafeOBJ variable for the hidden sort `Sys` of the OTS. Operator `try` is defined with the following equations:

```
-- for try
op c-try : Sys Pid -> Bool
eq c-try(S,I)
   = (pc(S,I) = l2 and top(queue(S)) = I) .
--
ceq pc(try(S,I),J)
    = (if I = J then cs
                else pc(S,J) fi)  if c-try(S,I) .
ceq queue(try(S,I)) = queue(S)    if c-try(S,I) .
ceq try(S,I) = S              if not c-try(S,I) .
```

`c-try(S,I)` denotes the effective condition of the transition `try`, which checks whether process `I`'s label is `l2` and the top element of the queue is equal to `I`. If the effective condition is satisfied, the transition `try` will be executed, and the execution will change the return value of observer `pc` to `cs` if the two processes `I` and `J` are the same. The execution of transition `try` does not change the return value of observer `queue`. If the effective condition does not hold, the state is not changed, which is described by the last equation. The other two operators `want` and `exit` could be defined with CafeOBJ equations in a similar way, which are not shown here.

In the OTS/CafeOBJ method, the verification of invariant properties is mainly done by structural induction, which means that what we need to do is to show firstly that the predicate to be proven invariant holds on any initial state (called the base case), and then to show that the predicate is preserved by execution of all transitions of the OTS (called the inductive case). In each inductive case, the case is usually split into multiple sub-cases with basic predicates (equations) declared in the CafeOBJ specification.

B. Maude Model Checker

Maude [4] is a high-performance language and system supporting both equational and rewriting logic computation for a wide range of applications. An important feature of Maude is that it has model checking facilities such as the `search` command and the Maude LTL model checker.

The basic units of Maude specifications are modules. There are two kinds of modules: functional modules and system modules. Maude functional modules define data types and operations on them by means of equational theories. System modules specify the initial model of a rewrite

Copyright (c) IARIA, 2012. ISBN: 978-1-61208-233-2

theory, which are essentially transition systems. A rewrite specification has rule statements: `crl` [*Label*] T_1 => T_2 `if` C_1 /\ C_2 /\ ... /\ C_k, in addition to the contents of functional modules. The condition part can be omitted if it is `true`. For a *finite* system, Maude `search` command explores all possible execution paths from the starting term (that represents an initial state) for reachable states satisfying some property.

C. A Specification Translation Method

We have proposed in [12] a way of translating CafeOBJ specifications for OTSs (the OTS/CafeOBJ specifications) into Maude specifications of a kind of rewriting transition systems for Bounded OTSs (the RWTS/Maude specifications). Bounded OTSs are the extension of OTSs to make it possible for the model checkers to explore a finite reachable state space of an OTS for counterexamples. To express the OTS/CafeOBJ expressible invariant properties in Maude forms, we have also proposed a simple way to generate Maude `search` commands from OTS/CafeOBJ formulas for invariant properties. We have proved that the proposed way of translation is sound with respect to counterexamples, namely that for any counterexample reported by Maude model checkers for the translated RWTS/Maude specifications, there exists a corresponding one in the original OTS/CafeOBJ specifications. We refer readers to [12] for translation details.

III. The Procedure of Combined Falsification and Verification (CFV)

In this section, we first give a brief review of the procedure Induction Guided Falsification (IGF) [5], and then introduce our proposed procedure – Combined Falsification and Verification (CFV), an extension of IGF.

A. A Review of Induction Guided Falsification (IGF)

As mentioned above, in the OTS/CafeOBJ method, although some invariant properties may be proved by rewriting and/or case splitting only, the generally used verification technique for proving invariant properties is structural induction [3]. The general procedure of structural induction is that: first, checking the base case, to show whether the state predicate to be proven invariant holds on any initial state, and second, checking the inductive case, to show whether the state predicate is preserved by the execution of any transition of the system. During proving the inductive case, we may have to discover and use other state predicates (called auxiliary state predicates) to strengthen the inductive hypothesis. Finding suitable state predicates to strengthen the inductive cases may be the most critical and difficult part of formal verification using theorem proving.

Structural induction works well when a state predicate to be proven invariant is indeed an invariant. However, it is quite often that we are trying to prove some state predicates that are essentially not invariants. Following structural induction, the usual way to know that a state predicate p under proving is not an invariant, is to show that p does not hold on any initial state, or to find some auxiliary state predicate, which is needed to prove p, but does not hold on any initial state. However, to find such an auxiliary state predicate, a lot of proof efforts are usually needed to manifest the problem. Such proof efforts can be extremely painful. Thus it is preferable that there exists some way, by which finding out errors lurking in the specifications can be easier and as earlier as possible.

Induction Guided Falsification (IGF) is a procedure that can reveal logical errors lurking in the specifications for theorem proving (falsification) as early as possible by employing model checking during the inductive verification of invariant properties, and the inductive verification can be used to reduce the state space needed for model checking to search a counterexample. The key concept that IGF lies on is *necessary lemmas*, which are obtained by applying *effective case splits*.

Definition 1: Effective case splits and Necessary lemmas. Consider proving a state predicate p to be invariant (i.e., to show $p(\upsilon)$ holds in any reachable state $\upsilon \in \mathcal{R}_{\mathcal{S}}$) by structural induction on the set of all reachable states $\mathcal{R}_{\mathcal{S}}$. In an inductive case where a transition $\tau_{y_1,\ldots,y_n}$ is taken into account, basically all we have to do is to prove $P(\upsilon_c, c_{l_1}, \ldots, c_{l_\alpha}) \Rightarrow P(\tau_{c_1,\ldots,c_n}(\upsilon_c), c_{l_1}, \ldots, c_{l_\alpha})$, where υ_c is a constant denoting an arbitrary state and each c_k is a constant denoting an arbitrary value of data type D_k. We suppose that a proposition $q_1 \vee \ldots \vee q_L$ is a tautology, where each q_l is in the form $Q_l(\upsilon_c, c_1, \ldots, c_n, c_{l_1}, \ldots, c_{l_\alpha})$. The case characterized by q_l is called a sub-case with respect to the inductive case. If the truth value of $P(\upsilon_c, c_{l_1}, \ldots, c_{l_\alpha}) \Rightarrow P(\tau_{c_1,\ldots,c_n}(\upsilon_c), c_{l_1}, \ldots, c_{l_\alpha})$ can be determined assuming each q_l, then $q_1 \vee \ldots \vee q_L$ is called an *effective case split* for this inductive case. Moreover, if the truth value is false, then $\forall \upsilon : \mathcal{R}_{\mathcal{S}}. \forall y_1 : D_1, \ldots, y_n : D_n, \forall x_{l_1} : D_{l_1}, \ldots, x_{l_\alpha} : D_{l_\alpha}. \neg Q_l(\upsilon, y_1, \ldots, y_n, x_{l_1}, \ldots, x_{l_\alpha})$ is called a *necessary lemma* of $p(\upsilon)$.

Note that this necessary lemma can surely make the inductive case `true`. If this necessary lemma is an invariant, then it means that the arbitrary state characterized by the sub-case is not reachable, and thus the false case is discharged and p is possibly an invariant; otherwise if this necessary lemma is not an invariant, then it means that the arbitrary state characterized by the sub-case is reachable, and thus p is not an invariant.

The procedure IGF is constructed based on two lemmas as its theoretical foundations. In the following, let $q(\upsilon)$ be $\forall y_1 : D_1, \ldots, y_n : D_n, \forall x_{l_1} : D_{l_1}, \ldots, x_{l_\alpha} : D_{l_\alpha}. \neg Q(\upsilon, y_1, \ldots, y_n, x_{l_1}, \ldots, x_{l_\alpha})$, and let q_l be $Q(\upsilon_c, c_1, \ldots, c_n, c_{l_1}, \ldots, c_{l_\alpha})$ where υ_c is a constant denoting an arbitrary state and each c_k is a constant

Copyright (c) IARIA, 2012. ISBN: 978-1-61208-233-2

denoting an arbitrary value of D_k.

Lemma 1: Let $\forall \upsilon : \mathcal{R}_{\mathcal{S}}.\ q(\upsilon)$ be a necessary lemma of $\forall \upsilon : \mathcal{R}_{\mathcal{S}}.\ p(\upsilon)$. If there exists a counterexample $ce_q \in \mathcal{CX}_{\mathcal{S},q}$ and $depth(ce_q) = N$, then (1) $ce_q \in \mathcal{CX}_{\mathcal{S},p}$, or (2) there exists a counterexample $ce_p \in \mathcal{CX}_{\mathcal{S},p}$ such that $depth(ce_p) = N + 1$.

Lemma 2: If $\mathcal{CX}_{\mathcal{S},p}$ is not empty and $depth(ce^{min}_{\mathcal{S},p}) = N + 1$, then there exists a necessary lemma $\forall \upsilon : \mathcal{R}_{\mathcal{S}}.q(\upsilon)$ of $\forall \upsilon : \mathcal{R}_{\mathcal{S}}.p(\upsilon)$ such that $\mathcal{CX}_{\mathcal{S},q}$ is not empty and $depth(ce^{min}_{\mathcal{S},q}) = N$.

Following the theories described in the above two lemmas, especially in lemma 2, we know that if a state predicate p to be proven invariant has counterexamples, then we can surely find and systematically construct some necessary lemmas. In turn, to prove these constructed necessary lemmas, if they do hold on any initial states, it is surely that we can find and systematically construct other necessary lemmas, and so on. As with this recursive process goes on, the depths of counterexamples of these necessary lemmas decrease. And from Lemma 1, we can conclude that if counterexamples exist for some necessary lemma, then p has counterexamples. This relieves us from traversing all needed necessary lemmas until we found one does not hold on any initial state.

Definition 2: Procedure IGF.
Input: an OTS and a state predicate p to be proven invariant.
Output: *Success* or *Fail*.

1. $\mathcal{P} := \{p\}$ and $\mathcal{Q} := \emptyset$.
2. Repeat the following until $\mathcal{P} = \emptyset$.
  (a) Choose a state predicate q from $\mathcal{P}$ and $\mathcal{P} := (\mathcal{P} - \{q\})$, where $q \in \textit{min-level}(\mathcal{P})$.
  (b) **Model checking** q in a finite reachable state space. If found a counterexample, terminate and return *Fail*.
  (c) Prove $\forall \upsilon_{\text{init}} : \mathcal{I}.\ q(\upsilon_{\text{init}})$. If it reduces to false, terminate and return *Fail*.
  (d) Find a set $\mathcal{G}$ of **necessary lemmas** such that $\forall \upsilon.[((\bigwedge_{g \in \mathcal{G}} g(\upsilon)) \wedge q(\upsilon)) \Rightarrow \forall \tau_{y_1,\dots,y_n} q(\tau_{y_1,\dots,y_n}(\upsilon))]$ reduces to true.
  (e) $\mathcal{Q} := \mathcal{Q} \cup \{q\}$ and $\mathcal{P} := \mathcal{P} \cup (\mathcal{G} - \mathcal{Q})$.
3. Terminate and return *Success*.

The basic idea of the procedure IGF is that: whenever trying to prove a state predicate, which is either the state predicate concerned (say p) or a constructed necessary lemma, model checking it first. Since model checking only checks a finite reachable state space, we use structural induction to prove p even if model checking did not find any counterexample. The falsifying and verifying is conducted in a breadth-first order with respect to the proof tree (to be introduced later), which is guaranteed by selecting a state predicate of minimal level in each loop described in step 2.(a).

B. The Algorithm of CFV

We have proved in [5] that IGF is sound and complete with respect to falsification, and is sound but not complete with respect to verification. This implies that IGF may work well for proving a state predicate with counterexamples, namely that for falsifying it. But in the situation that a given state predicate is indeed an invariant (no counterexample), the procedure may not terminate and successfully prove the state predicate due to using necessary lemmas as the only way to strengthen inductive hypothesis.

As an extension of IGF for enhancing the verification capability, Combined Falsification and Verification (CFV) is a more general procedure that aims at both falsification and verification. The main difference between the procedures IGF and CFV lies on using what kind of lemmas to strengthen the inductive hypothesis. In the procedure IGF, we always construct and use necessary lemmas to strengthen inductive hypothesis, but in the procedure CFV, we systematically use some other stronger lemmas (we say a state predicate p is stronger than another q if $p \Rightarrow q$) that may be more simple and appropriate, and until no such stronger lemmas suffice to strengthen inductive hypothesis, the necessary lemmas are used at last, which are the weakest lemmas.

The algorithm of the procedure CFV is shown in Definition 3. Basic idea of the procedure CFV is almost same as the procedure IGF. But since the state predicates used to strengthen the inductive hypothesis are sometimes not necessary lemmas, we need to consider more (rather than directly concluding that the state predicate concerned is not an invariant, as done in IGF) when a counterexample is reported for a state predicate, or the state predicate does not hold on any initial state, because in both cases, what we know is only that the state predicate itself is not an invariant.

The key operation or function in the procedure CFV is $process$. When a counterexample is found by model checking for a state predicate, or the state predicate does not hold on any initial state, operation $process$ is called and it returns different values according to the category of the state predicate. The possible output of operation $process$ is either F, which means the procedure CFV should be terminated and return Fail; or (X,Y), where X denotes a set of state predicates that are not appropriate or correct and should be removed from $\mathcal{P}$ and $\mathcal{Q}$, and Y denotes a set of state predicates that are possibly appropriate or correct and should be added to $\mathcal{P}$.

The primary part (except the details of the operation $process$) of the verification procedure CFV can be represented as a flow chart as shown in Figure 1.

Copyright (c) IARIA, 2012.

We now explain the basic idea of the procedure CFV by using some examples. Assume a tree-like structure shown in Figure 2.(a) that represents the proof of a state predicate p. The tree structure is rooted, unordered, and labeled. The tree is supposed to be constructed using a breadth-first manner. The root of the tree is p, and all the other offspring nodes are constructed lemmas (state predicates) to strengthen certain inductive hypothesis for proving their respective parent nodes (state predicates), where the nodes with superscript n are necessary lemmas and those without superscript n are not necessary lemmas.

Assume that we find a counterexample for state predicate z by model checking, or z does not hold on any initial state, which means that z is not an invariant. Since z is not a necessary lemma (without the superscript n), the procedure CFV will then use a systematical way (to be introduced later) to generate and use another state predicate, say s instead of z, to strengthen an inductive hypothesis (characterized by the label l_8) to prove r_2^n, which is shown in Figure 2.(b). And the state predicate z (and also all its children nodes, if any) will be removed from $\mathcal{P}$ and $\mathcal{Q}$, and s will be added to $\mathcal{P}$.

We now assume that the state predicate z is a necessary lemma (denoted by z^n shown in Figure 3.(a)), and we know z is not an invariant by either model checking or checking any initial state, then the procedure CFV will try to find, in its parent list, the nearest state predicate to z^n that is not a necessary lemma (assume this state predicate is x), and try to generate and use other lemmas, instead of x, to strengthen the inductive hypothesis denoted by $label(x)$. Let us see the example in Figure 3.(a), since the nearest state predicate to z^n that is not a necessary lemma is q_2, then we know that q_2 should be replaced by some other state predicates. Note that since q_2 is also used to strengthen the inductive hypothesis characterized by l_4 to prove q_1^n, then the procedure will construct two lemmas, say s_1 and s_2, to strengthen the inductive hypothesis characterized by l_4 and l_2, which is shown in Figure 3.(b). The state predicate q_2 and all its recursive children nodes should be removed from $\mathcal{P}$ and $\mathcal{Q}$, and the two newly constructed state predicate s_1 and s_2 will be added to $\mathcal{P}$.

As another example, let us see Figure 4 (see below). If we find a counterexample for the state predicate z^n, or z^n does not hold on any initial state. Since z^n is a necessary lemma, and there exists a parent list of z^n, say $r_2^n\ q_2^n$, where any state predicate in this list is a necessary lemma. Then the procedure CFV is terminated and returns Fail, which means that the state predicate p to be proven invariant is not an invariant. This situation is exactly same as the procedure IGF, which is based on the theory defined in Lemma 1.

After introducing the procedure CFV, another thing left unexplained is how the procedure *systematically* constructs other state predicates when a state predicate, which is not a necessary lemma, is not appropriate, as mentioned in the above examples. To explain this, let us first see more exactly what is the form of a sub-case. The sub-case is characterized by a set of equations, say E. When CafeOBJ system reduces to false for this sub-case, a necessary lemma in the form $\neg(\bigwedge_{e \in E} e)$ can be constructed. Note that from this set of equations E, we can also systematically construct other state predicates, and each such state predicate is in the form $\neg(\bigwedge_{e' \in E'} e')$, where $E' \in 2^E$, and these state predicates are stronger than the necessary lemma since $\neg(\bigwedge_{e' \in E'} e') \Rightarrow \neg(\bigwedge_{e \in E} e)$. Basically, all of these state predicates are candidates that can be used, instead of the necessary lemma, to strengthen inductive hypothesis. However, only if they satisfy two conditions, they become the real candidates that will be used by the procedure CFV. The first condition is of course they should be able to make the inductive case `true`; and the second condition is that by model checking them, no counterexample should be found.

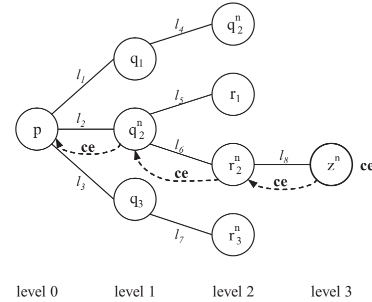

Figure 4. The third sample tree of proving p with CFV

Let us consider proving $\forall v : \mathcal{R}_{\mathcal{S}}.\ p(v)$ is an invariant. In an inductive case denoted by a transition $\tau_{y_1,\ldots,y_n}$, CafeOBJ system returns false for a sub-case characterized by a set of equations e_1, e_2, e_3, e_4. Then all the lemmas we can construct from these equations are shown below:

One equation	$\neg e_1$, $\neg e_2$, $\neg e_3$, $\neg e_4$
Two equations	$\neg(e_1 \wedge e_2)$, $\neg(e_1 \wedge e_3)$, $\neg(e_1 \wedge e_4)$ $\neg(e_2 \wedge e_3)$, $\neg(e_2 \wedge e_4)$, $\neg(e_3 \wedge e_4)$
Three equations	$\neg(e_1 \wedge e_2 \wedge e_3)$, $\neg(e_1 \wedge e_2 \wedge e_4)$ $\neg(e_1 \wedge e_3 \wedge e_4)$, $\neg(e_2 \wedge e_3 \wedge e_4)$
Four equations	$\neg(e_1 \wedge e_2 \wedge e_3 \wedge e_4)$

After the procedure CFV filtered some of them according to the two conditions, CFV will use the remaining lemmas to strengthen the inductive cases in an order from “One equation” to “Four equations”, and the “Four equations” lemma is the necessary lemma.

IV. Towards Formal Analysis of HSTM Designs with the Procedure CFV

Hierarchical State Transition Matrix (HSTM) [13] is a table based modeling language for developing designs of software systems. A HSTM design, namely a design developed with HSTM, consists of multiple STMs organized in a hierarchical structure. Each STM models a component of the design in the form of a table and specifies behaviors

Copyright (c) IARIA, 2012. ISBN: 978-1-61208-233-2

of the component when certain events are dispatched in certain states. A simple sample STM is shown below for demonstration purpose. The informal meaning of the STM is that: the STM has two states $S1$ and $S2$; there are two events $e1$ and $e2$ that may happen to the STM; if, for example, $e1$ is dispatched when the STM is in states $S1$, $action_1$ will be executed and then the STM switches to states $S2$. The other cells have similar meanings.

	$S1$	$S2$
$e1$	$S2$	$S1$
	$action_1$	$action_2$
$e2$	$S2$	$S1$
	$action_3$	$action_4$

HSTM has been widely accepted and used by particularly Japanese embedded software industry, and has been adopted as the modeling language of commercial model-based tools such as ZIPC [10]. However, despite of its popularity, there is still lack of mechanized formal verification supports for conducting rigorous and automatic analysis to improve reliability of HSTM designs. Based on this need, we have been developing a HSTM model checker called Garakabu2 [8], [9].

Garakabu2 implements SMT-based [6] Bounded Model Checking (BMC) [7] algorithms for verification of HSTM designs. In addition, specific considerations for its practical usabilities for non-experts in formal methods have been taken into account during its development. However, one issue is that Garakabu2 only conducts falsification due to that only a bounded state space is checked with BMC. This is sufficient for normal users like software engineers who wish to explore bugs in HSTM designs. But for expert users like those who have sufficient knowledge on inductive theorem proving, it may be desirable that verification functionality (i.e., proving correctness) is also available in Garakabu2.

Due to the fact that each STM is essentially a state transition system and a HSTM is just a set of STM organized in a hierarchical structure, it is possible that a HSTM design could be represented with OTS and thus be formally analyzed with the procedure CFV. One key issue in using CFV to analyze HSTM designs is to translate a HSTM design into an OTS (which is to be specified in CafeOBJ specification). This is not difficult since a parser for HSTM designs has been implemented in Garakabu2 and is ready to be used. We have been formalizing the translation rules from HSTM designs into OTSs and the details will be reported in another opportunity.

V. Conclusions and Future Work

In this paper, we first briefly reviewed the procedure IGF [5], and then described our proposed procedure CFV, which is an extension of IGF for both falsification and verification of systems specified in CafeOBJ specification (with OTS as the background concept). Note that although the proposed procedure CFV relies on some specific features of the OTS/CafeOBJ method and Maude model checkers, it may be revised and extended to combinations of inductive verification techniques of other theorem proving and other model checking techniques while remaining the basic idea of the procedure.

Furthermore, we simply investigated the possibility of applying the procedure CFV to formal analysis of HSTM designs. We have been formalizing translation rules from HSTM designs into OTSs. In the future, we plan to implement this translation in Garakabu2, and implement interfaces to connect Garakabu2 with CafeOBJ and Maude systems, by which Garakabu2 could be used by formal methods experts for conducting formal analysis of HSTM designs with both interactive theorem proving and model checking by following the CFV procedure.

References

[1] J. Mseguer, M. Palomino, and N. Martí-Oliet, “Equational abstractions.” in *CADE*, 2003, pp. 2–16.

[2] E. Clarke, O. Grumberg, and D. Peled, *Model Checking*. MIT Press, 1999.

[3] K. Ogata and K. Futatsugi, “Proof Scores in the OTS/CafeOBJ Method,” in *FMOODS 2003*, ser. LNCS, vol. 2884. Springer, 2003, pp. 170–184.

[4] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott, *Maude 2.0 Manual: Version 2.1*, March 2004.

[5] K. Ogata, M. Nakano, W. Kong, and K. Futatsugi, “Induction-Guided Falsification,” in *ICFEM 2006*, ser. LNCS, vol. 4260. Springer, 2006, pp. 114–131.

[6] C. Barrett, R. Sebastiani, S. Seshia, and C. Tinelli, *Handbook of Satisfiability*. IOS Press, 2009, vol. 185, ch. 26, pp. 825–885.

[7] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, “Symbolic model checking without BDDs,” in *5th TACAS*. Springer, 1999, pp. 193–207.

[8] W. Kong, N. Katahira, M. Watanabe, T. Katayama, K. Hisazumi, and A. Fukuda, “Formal verification of software designs in hierarchical state transition matrix with SMT-based bounded model checking,” in *18th APSEC*. IEEE CS, 2011, pp. 81–88.

[9] W. Kong, T. Shiraishi, N. Katahira, M. Watanabe, T. Katayama, and A. Fukuda, “An SMT-based approach to bounded model checking of design in state transition matrix,” *IEICE Transactions on Information and Systems*, vol. E94-D(5), pp. 946–957, 2011.

[10] CATS Co., Ltd., Japan, “ZIPC v10,” www.zipc.com, [retrieved: October 2012].

[11] R. Diaconescu and K. Futatsugi, *CafeOBJ Report*, ser. AMAST Series in Computing. World Scientific, 1998, no. 6.

[12] W. Kong, K. Ogata, T. Seino, and K. Futatsugi, “A Lightweight Integration of Theorem Proving and Model Checking for System Verification,” in *APSEC 2005*. IEEE CS, 2005, pp. 59–66.

[13] M. Watanabe, “Extended hierarchy state transition matrix design method,” in *CATS Technical Report*, 1998.

Definition 3: Procedure CFV.
Input: an OTS and a state predicate p to be proven invariant.
Output: *Success* or *Fail*.

1. $\mathcal{P} := \{p\}$ and $\mathcal{Q} := \emptyset$.
2. Repeat the following until $\mathcal{P} = \emptyset$.
(1) Choose a state predicate q from $\mathcal{P}$ and $\mathcal{P} := (\mathcal{P} - \{q\})$, where $q \in min\text{-}level(\mathcal{P})$.
(2) Case [**Model checking** q in a finite reachable state space] *of:*
 (a) Counterexample: *case* [$process(p,q)$] *of:*
  (I) F, then terminate and returns *Fail*.
  (II) (X,Y), then $\mathcal{Q} := (\mathcal{Q} - X)$, $\mathcal{P} := ((\mathcal{P} - X) \cup (Y - \mathcal{Q}))$.
 (b) No counterexample, *case* [prove $\forall v_{\text{init}} : I.\ q(v_{\text{init}})$] *of:*
  (I) reduces to false, *case* [$process(p,q)$] *of:*
   (1°) *F*, then terminate and returns *Fail*.
   (2°) *(X,Y)*, then $\mathcal{Q} := (\mathcal{Q} - X)$, $\mathcal{P} := ((\mathcal{P} - X) \cup (Y - \mathcal{Q}))$.
  (II) reduces to true, then $\mathcal{G} := valid(q)$; $\mathcal{Q} := \mathcal{Q} \cup \{q\}$; $\mathcal{P} := \mathcal{P} \cup (\mathcal{G} - \mathcal{Q})$.
3. Terminate and return *Success*.

where:

process(m,n):
Input: two state predicates m and n.
Output: either F or a tuple *(X,Y)*, where X and Y are two sets of state predicates.

1. $X := \emptyset$ and $Y := \emptyset$.
2. Case [$n = m$] *of:*
(1) true, then terminate and return *F*.
(2) false, *case* [n is a *necessary lemma*] *of:*
 (a) true, then *case* [$(parentList(n) = \emptyset) \vee (\exists\, List \in parentList(n)$, where any node in $List$ is a necessary lemma)] *of:*
  (I) true, then terminate and return *F*.
  (II) false, then *For each* $List \in parentList(n)$ *do*
   (1°) $X := X \cup\ childrenSet(z)$, $Y := Y \cup\ tc\text{-}valid(previous(z), z)$, where z is the nearest state predicate in $List$ to n that is not a necessary lemma;
   (2°) return *(X,Y)*.
 (b) false, then
  (I) $X := \{n\} \cup childrenSet(n)$;
  (II) For all $z \in parent(n)$ *do*
   (1°) $Y := Y \cup\ tc\text{-}valid(z,n)$;
   (2°) return *(X,Y)*.

valid(m) $= \{\mathcal{G} \mid \forall v : \Upsilon.[((\bigwedge_{g\in\mathcal{G}}\ g(v)) \wedge m(v)) \Rightarrow \forall \tau_{y_1,\ldots,y_n} : \mathcal{T}.\ m(\tau_{y_1,\ldots,y_n}(v))]\}$.

tc-valid(m,n) $= n'$, where under the case denoted by $label(n)$, which includes the inductive case denoted by a transition $\tau_{y_1,\ldots,y_n} := lt(label(n))$, and a sub-case denoted by $lc(label(n))$, such that: $\forall v : \Upsilon.[n'(v) \wedge m(v) \Rightarrow m(\tau_{y_1,\ldots,y_n}(v))]$ reduces to true.

Copyright (c) IARIA, 2012. ISBN: 978-1-61208-233-2

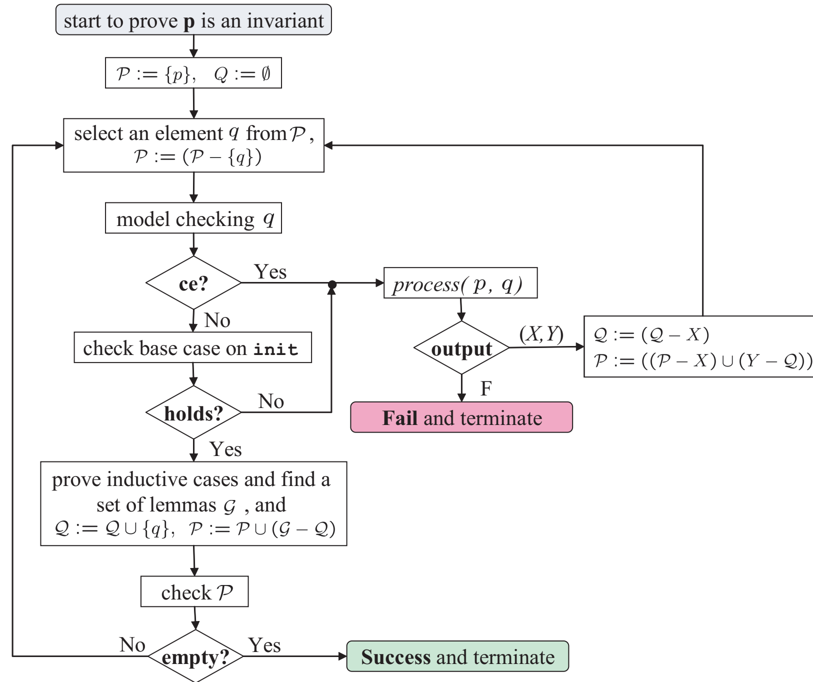

Figure 1. Flow chart representation of the procedure CFV

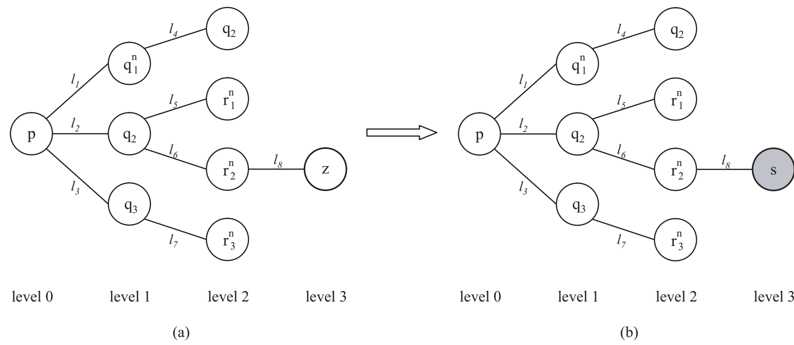

Figure 2. A sample tree of proving state predicate p with CFV

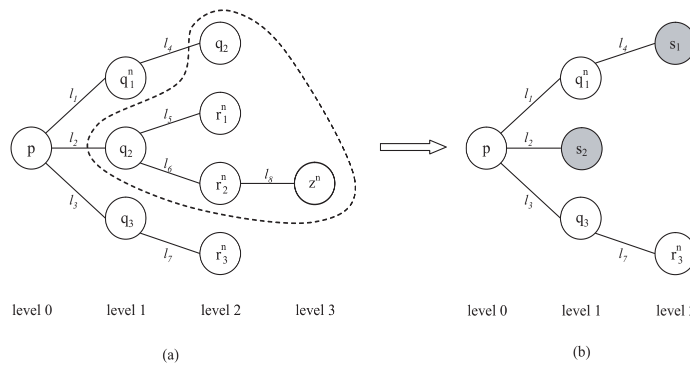

Figure 3. Another sample tree of proving state predicate p with CFV