A Combination of Forward & Backward Reachability Analysis Methods

Kazuhiro Ogata and Kokichi Futatsugi

School of Information Science, JAIST
{ogata, kokichi}@jaist.ac.jp

Abstract. Induction-guided falsification (IGF) is a combination of bounded model checking (BMC) and structural induction, which can be used for falsification of invariants. IGF can also be regarded as a combination of forward and backward reachability analysis methods. This is because BMC is a forward reachability analysis method and structural induction can be regarded as a backward reachability analysis method. We report on a case study in which a variant of IGF has been used to systematically find a counterexample showing that NSPK does not enjoy the agreement property.

Keywords: agreement property, CafeOBJ, bounded model checking, falsification, NSPK, structural induction, Maude

1 Introduction

Bounded model checking (BMC)[1] has been used to discover a counterexample showing that a hardware or software system does not enjoy a safety property. It (or its concept) has been adopted by some software analysis tools such as Alloy[2]. Basically it starts with some initial states of a system and exhaustively traverses the state space reachable from the initial states up to some specific depth. Therefore, BMC is a forward reachability analysis method.

A backward reachability analysis method starts with some states of a system such that a safety property is broken and traverses the state space reachable in a backward sense from the states. If it reaches an initial state of the system, the system does not enjoy the safety property. If the entire state space reachable in a backward sense from such an arbitrary state does not contain any initial states, the system enjoys the safety property. Among tools adopting a backward reachability analysis method is Maude-NPA[3].

We have proposed a combination of BMC and structural induction called induction-guided falsification (IGF)[4]. IGF uses as BMC the search functionality provided by CafeOBJ[5] and Maude[6]. In IGF, structural induction is conducted by human users by writing what are called proof scores in CafeOBJ. IGF can also be regarded as a combination of forward and backward reachability analysis methods. This is because structural induction can also be regarded as a backward reachability analysis method.

In this paper, we review IGF and report on a case study in which a variant of IGF, where non-necessary lemmas may be used, has been used to systematically discover a counterexample showing that NSPK[7] does not enjoy the agreement property. Many case studies have been reported, systematically discovering a counterexample showing that NSPK does not enjoy the nonce secrecy property[8, 9, 3]. To our best knowledge, however, few case studies have been reported for the agreement property.

The rest of the paper is organized as follows. §2 describes the OTS/CafeOBJ method[10] for specification and verification of systems. A simple example is used to describe the method and demonstrate that (structural) induction can be used to falsify that a system enjoys a property. §3 describes the search functionality. §4 describes a viewpoint that regards (structural) induction as a backward reachability analysis method, reviews IGF, and introduces a variant of IGF. §5 reports on the case study. §6 mentions some related work. §7 concludes the paper.

2 The OTS/CafeOBJ Method

2.1 Observational Transition Systems

We suppose that there exists a universal state space denoted by Υ and that each data type used in OTSs is provided. The data types include Bool for Boolean values. A data type is denoted by D with a subscript such as D_{o1} and D_o.

Definition 1 (OTSs). *An observational transition system (OTS) consists of*

- $\mathcal{O}$ *: A set of observers. Each observer is a function* $o : \Upsilon\ D_{o1} \ldots D_{om} \rightarrow D_o$*. The equivalence between two states* υ_1, υ_2 *(denoted as* $\upsilon_1 =_{\mathcal{S}} \upsilon_2$*) is defined with respect to (wrt) values returned by the observers.*
- $\mathcal{I}$ *: The set of initial states such that* $\mathcal{I} \subseteq \Upsilon$*.*
- $\mathcal{T}$ *: A set of transitions. Each transition is a function* $t : \Upsilon\ D_{t1} \ldots D_{tn} \rightarrow \Upsilon$*. Each transition* t*, together with any other parameters* $y_1, \ldots, y_n$*, preserves the equivalence between two states. Each* t *has the effective condition* $c\text{-}t : \Upsilon\ D_{t1} \ldots D_{tn} \rightarrow \text{Bool}$*. If* $\neg c\text{-}t(\upsilon, y_1, \ldots, y_n)$*, then* $t(\upsilon_1, y_1, \ldots, y_n) =_{\mathcal{S}} \upsilon$*.*

Given an OTS $\mathcal{S}$ and a state υ, $t(\upsilon, y_1, \ldots, y_n)$ is called a successor state of υ wrt $\mathcal{S}$ for any $y_1, \ldots, y_n$. We may omit "wrt $\mathcal{S}$" if it is clear from the context.

Definition 2 (Reachable States). *Given an OTS* $\mathcal{S}$ *and a set* $\mathbf{U}$ *of states, the reachable states from* $\mathbf{U}$ *wrt* $\mathcal{S}$ *are inductively defined as follows:*

- *Each state in* $\mathbf{U}$ *is reachable from* $\mathbf{U}$*.*
- *If a state* $\upsilon \in \Upsilon$ *is reachable from* $\mathbf{U}$*, so is* $t(\upsilon, y_1, \ldots, y_n)$ *for each* $t \in \mathcal{T}$ *and any other parameters* $y_1, \ldots, y_n$*.*

We may omit "wrt $\mathcal{S}$" and write "υ is reachable from $\mathbf{U}$" if $\mathcal{S}$ is clear from the context. When $\mathbf{U}$ is $\mathcal{I}$, we may omit "from $\mathcal{I}$" and write "υ is reachable wrt $\mathcal{S}$" or "υ is reachable".

Let $\mathcal{R}_{\mathcal{S},\mathbf{U}}$ be the set of all states reachable from $\mathbf{U}$ wrt $\mathcal{S}$. $\mathcal{R}_{\mathcal{S},\mathbf{U}}$ may be called the state space reachable from $\mathbf{U}$. Let $\mathcal{R}_{\mathcal{S}}$ be $\mathcal{R}_{\mathcal{S},\mathcal{I}}$. $\mathcal{R}_{\mathcal{S}}$ may be called the reachable state space. When $\mathbf{U}$ is a singleton, say $\{u\}$, we may write $\mathcal{R}_{\mathcal{S},u}$,

$\upsilon \in \mathcal{R}_{\mathcal{S},u}$ is called reachable from u and u is called backward-reachable from $\upsilon \in \mathcal{R}_{\mathcal{S},u}$. Given an OTS $\mathcal{S}$ and two states $\upsilon_1, \upsilon_2 \in \Upsilon$, the depth from υ_1 to υ_2 wrt $\mathcal{S}$ ($\mathrm{depth}_{\mathcal{S}}(\upsilon_1, \upsilon_2)$) is 0 if $\upsilon_2 =_{\mathcal{S}} \upsilon_1$ and $d+1$ if $\mathrm{depth}_{\mathcal{S}}(\upsilon_1, \upsilon_3) = d$ and υ_2 is a successor state of υ_3. If υ_2 is not reachable from υ_1, $\mathrm{depth}_{\mathcal{S}}(\upsilon_1, \upsilon_2)$ is ∞. Let $\mathcal{R}^{\leq d}_{\mathcal{S},u}$ be $\{\upsilon \in \mathcal{R}_{\mathcal{S},u} \mid \mathrm{depth}_{\mathcal{S}}(u, \upsilon) \leq d\}$.

Definition 3 (Invariants). *Given an OTS $\mathcal{S}$, a state predicate $p : \Upsilon \to \mathrm{Bool}$ is an invariant wrt $\mathcal{S}$ if $(\forall \upsilon \in \mathcal{R}_{\mathcal{S}})\, p(\upsilon)$.*

We may omit "wrt $\mathcal{S}$" and write "p is an invariant" if $\mathcal{S}$ is clear from the context.

CafeOBJ, an algebraic specification language, is used to specify OTSs. Υ is denoted by a sort, say `Sys`. Each $o \in \mathcal{O}$ is denoted by an operator (called an observation operator) declared as follows: "`op o : Sys` $\mathtt{D}_{o1} \ldots \mathtt{D}_{om}$ `->` $\mathtt{D}_o$", where each $\mathtt{D}_*$ is a sort corresponding to D_*.

An arbitrary initial state in $\mathcal{I}$ is denoted by an operator declared as follows: "`op init : -> Sys {constr}`". Operators with no arguments such as `init` are called constants. For each $o \in \mathcal{O}$, declared is an equation "`eq o(init,`$\mathtt{X}_1$`,...,`$\mathtt{X}_m$`) = f(`$\mathtt{X}_1$`,...,`$\mathtt{X}_m$`) .`", where each $\mathtt{X}_*$ is a CafeOBJ variable of sort $\mathtt{D}_*$ and `f(`$\mathtt{X}_1$`,...,`$\mathtt{X}_m$`)` is a term denoting the value returned by o, together with any other parameters, in an arbitrary initial state. Note that each CafeOBJ variable occurring in an equation (or a transition rule; see §3) is universally quantified and its scope is in the equation (or the transition rule).

Each $t \in \mathcal{T}$ is denoted by an operator (called a transition operator) declared as follows: "`op t : Sys` $\mathtt{D}_{t1} \ldots \mathtt{D}_{tn}$ `-> Sys {constr}`". For each `o` and `t`, a conditional equation is declared: "`ceq o(t(S,`$\mathtt{Y}_1$`,...,`$\mathtt{Y}_n$`),`$\mathtt{X}_1$`,...,`$\mathtt{X}_m$`) = o-t(S,`$\mathtt{Y}_1$`,...,`$\mathtt{Y}_n$`,`$\mathtt{X}_1$`,...,`$\mathtt{X}_m$`) if c-t(S,`$\mathtt{Y}_1$`,...,`$\mathtt{Y}_n$`) .`", where `c-t(S,...)` corresponds to $c\text{-}t(\upsilon,\ldots)$ and `o-t(S,...)` does not use any transition operators. The equation says how t changes the value observed by o if the effective condition holds. If `o-t(S,...)` is always the same as `o(S,`$\mathtt{X}_1$`,...,`$\mathtt{X}_m$`)`, the condition may be omitted.

For each `t`, one more conditional equation is declared: "`ceq t(S,`$\mathtt{Y}_1$`,...,`$\mathtt{Y}_n$`) = S if not c-t(S,`$\mathtt{Y}_1$`,...,`$\mathtt{Y}_n$`) .`", which says that t changes nothing if the effective condition does not hold.

As indicated by `{constr}`, `init` and each `t` are constructors of sort `Sys`[1]. They construct $\mathcal{R}_{\mathcal{S}}$.

A simple example is used to describe OTSs. The example used is a flawed mutual exclusion protocol.

Example 1 (Flawed Mutex Protocol). Multiple processes share a Boolean variable *locked* whose initial value is false. Each process executes the pseudo-program:

```
Loop: "Remainder Section"
  rs: wait until $locked$ = false;
  es: $locked$ := true;
```

[1] Sort `Sys` denotes $\mathcal{R}_{\mathcal{S}}$ but not Υ if the constructor-based logic[11] is adopted, which is the current underlined logic of the OTS/CafeOBJ method.

        “Critical Section”
    cs: *locked* := false;

Initially each process is in Remainder Section (RS). If a process wants to enter Critical Section (CS), it waits at label rs until *locked* becomes false and then sets *locked* to true at label es before entering CS. When it leaves CS, it sets *locked* to false at label cs and then goes back to RS.

How to specify an OTS $\mathcal{S}_{\mathrm{FMP}}$ formalizing the protocol is described. Two observers are used. The corresponding observation operators are as follows: “`op locked : Sys -> Bool`” and “`op pc : Sys Pid -> Label`”, where sort `Pid` denotes process identifiers (IDs) and sort `Label` denotes the labels rs, es and cs. `locked` returns the value of *locked* in a given state, and `pc` returns the label at which a given process is in a given state.

In the rest of this section, let `S`, `I` and `J` be CafeOBJ variables of sorts `Sys`, `Pid` and `Pid`, respectively. The values returned by the two observers in an arbitrary initial state denoted by `init` are specified as follows: “`eq locked(init) = false .`” and “`eq pc(init,I) = rs .`”.

Three transitions are used. The corresponding transition operators are as follows: “`ops try enter exit : Sys Pid -> Sys` {`constr`}”. `try`, `enter`, and `exit` correspond to one iteration of the loop at label rs, the assignment at label es, and the assignment at label cs, respectively.

The set of equations specifying how `try` changes the values observed by the two observers is as follows:

```
eq locked(try(S,I)) = locked(S) .
ceq pc(try(S,I),J)
 = (if I = J then es else pc(S,J) fi) if c-try(S,I) .
ceq try(S,I) = S if not c-try(S,I) .
```

where `c-try(S,I)` is defined as `pc(S,I) = rs and not locked(S)`. `enter` and `exit` are defined likewise. Let `MUTEX` be a module in which $\mathcal{S}_{\mathrm{FMP}}$ is specified.

2.2 Falsification by Structural Induction

Verification of invariants is conducted by writing proof scores in CafeOBJ and executing them with the CafeOBJ system. Verification that a state predicate is an invariant wrt $\mathcal{S}_{\mathrm{FMP}}$ is used as an example to describe how to write proof scores in CafeOBJ. The state predicate used is $(\forall \mathtt{I}, \mathtt{J} : \mathtt{Pid})$ `inv1(S,I,J)`, where `inv1(S,I,J)` is `pc(S,I) = cs and pc(S,J) = cs implies I = J`. The predicate formalizes what is called the mutual exclusion property. Let `MUTEX-PREDS` be a module in which `MUTEX` is imported (namely that it is available) and state predicates to verify such as `inv1` are specified.

Verification starts with use of the structural induction on $\mathcal{R}_{\mathcal{S}_{\mathrm{FMP}}}$ (or sort `Sys`). Then, we have four CafeOBJ code fragments because there are the four constructors. Two out of the four CafeOBJ code fragments enclosed with commands `open` and `close` are as follows:

```
open MUTEX-BASE               open MUTEX-ISTEP
  red inv1(init,i,j) .          eq s' = enter(s,k) .  red istep1 .
close                         close
```

MUTEX-BASE is a module in which MUTEX-PREDS is imported. s, s', i, j and k are constants declared in MUTEX-BASE. s is used to denote an arbitrary state, s' an arbitrary successor state of s, and i, j and k arbitrary process identifiers. MUTEX-ISTEP is a module in which MUTEX-BASE is imported. istep1 is a constant declared in MUTEX-ISTEP. istep1 is defined as inv1(s,i,j) implies inv1(s',i,j). inv1(s',i,j) is the formula to prove in each induction case and inv1(s,i,j) is an instance of the induction hypothesis $(\forall \mathtt{I}, \mathtt{J} : \mathtt{Pid})$ inv1(s,I,J). Command open makes a temporary module in which a given module is imported, and command close destroys such a temporary module. Command red reduces a given term by regarding equations as left-to-right rewrite rules. The four CafeOBJ code fragments are the proof score in progress of inv1. CafeOBJ code fragments in proof scores (in progress as well) are called proof passages. The proof passage for init is for the base case, while the remaining three ones for the induction step, or the three induction cases.

If CafeOBJ returns true for a proof passage, the proof passage is discharged. If CafeOBJ returns true for each proof passage in the proof score of a predicate and all lemmas used in the proof score have been proved, the predicate has been proved, namely that it is an invariant wrt an OTS concerned.

CafeOBJ returns true for the proof passage for init and then the base case is discharged. Since CafeOBJ does not return true for the remaining three, however, we need to transform the proof passages with case splitting and lemma conjecture/use.

Let us take the induction case for enter. The proof passage is first transformed into two proof passages with case splitting based on the effective condition of enter. The two proof passages correspond to the two cases: (1) c-enter(s,k) = false, and (2) c-enter(s,k) = true. CafeOBJ returns true for the first case but not for the second case. Since c-enter(s,k) = true is equivalent to pc(s,k) = es, the latter can be used instead of the former. Even if so, CafeOBJ does not return true for the second case.

The proof passage is next transformed into four proof passages with case splitting based on the two propositions i = k and j = k found in the result returned by CafeOBJ. The four proof passages correspond to the four cases: (1) i = k, j = k, (2) (i = k) = false, (j = k) = false, (3) i = k, (j = k) = false, and (4) (i = k) = false, j = k. CafeOBJ returns true for the first two cases but not for the remaining two cases. Let us take the third case.

The corresponding proof passage is then transformed into two proof passages with case splitting based on the proposition pc(s,j) = cs. The two proof passages correspond to the two cases: (1) (pc(s,j) = cs) = false, and (2) pc(s,j) = cs. CafeOBJ returns true for the first case but false for the second case.

The proof passage corresponding to the second case is as follows:

```
open MUTEX-ISTEP
```

```
 eq pc(s,k) = es .    eq i = k .    eq (j = k) = false .
 eq pc(s,j) = cs .    eq s’ = enter(s,k) .    red istep1 .
close
```

If inv1 holds for $\mathcal{S}_{\mathrm{FMP}}$, then an arbitrary state s characterized by the first four equations in the proof passage is unreachable wrt $\mathcal{S}_{\mathrm{FMP}}$. Therefore, we can conjecture a lemma from the four equations to discharge the proof passage. If one of such equations such as i = k has a fresh constant as one side and CafeOBJ still returns false even after replacing all the occurrences of the fresh constant with the other side in the proof passage and deleting the equation, then we can use the remaining equations to conjecture a lemma.

For this proof passage, a lemma can be conjectured from the following three equations by basically conjoining the equations with conjunctions, negating the obtained formula, and replacing fresh constants with appropriate variables: eq pc(s,i) = es ., eq (j = i) = false ., and eq pc(s,j) = cs . The lemma is not(pc(S,I) = es and pc(S,J) = cs and not(I = J)), which is referred as inv2(S,I,J). This lemmas has the property that if inv1 holds for $\mathcal{S}_{\mathrm{FMP}}$, so does inv2. Or in contrapositive form, if inv2 does not hold for $\mathcal{S}_{\mathrm{FMP}}$, neither does inv1. Lemmas that have this property are called necessary lemmas of the original predicates[4]. inv2 is a necessary lemma of inv1 and only the lemma needed to discharge the proof score of inv1.

In the verification of inv2, we conjecture the two lemmas not(pc(S,I) = rs and pc(S,J) = cs and not(I = J) and not(locked(S))) and not(pc(S,I) = es and pc(S,J) = es and not(I = J)), which are referred as inv3(S,I,J) and inv4(S,I,J), respectively. Both inv3 and inv4 are necessary lemmas of inv2.

We only need inv1 as a lemma to discharge the proof score of inv3, but conjecture the following lemma for inv4: not(pc(S,I) = es and pc(S,J) = rs and not(I = J) and not(locked(S))), which is referred as inv5(S,I,J). inv5 is a necessary lemma of inv4.

In the verification that inv5 holds for $\mathcal{S}_{\mathrm{FMP}}$, the following lemma is conjectured: not(pc(S,I) = rs and pc(S,J) = rs and not(I = J) and not(locked(S))), which is referred as inv6(S,I,J). inv6 is a necessary lemma of inv5.

inv6(init,i,j) reduces to false if i is different from j, from which we can conclude that inv1 does not hold for $\mathcal{S}_{\mathrm{FMP}}$ because every lemma used is a necessary lemma of its original predicate. This example demonstrates that structural induction can also be used to falsify that a system enjoys an invariant.

3 Bounded Model Checking (BMC) of OTSs

Instead of a set of equations, a transition rule can also be used to specify each transition $t \in \mathcal{T}$ of an OTS $\mathcal{S}$. If so, the search functionality can be used. The search functionality is in the form:

red *init* =(n,d)=>* *pattern* suchThat *cond* .

where *init* is a ground term, *pattern* a state pattern, *cond* a Boolean term, and n and d natural numbers or `*` denoting the infinity. "suchThat *cond*" is an option. The search functionality exhaustively traverses $\mathcal{R}^{\leq d}_{\mathcal{S},init}$ in a breadth-first manner so as to find at most n states such that they match *pattern* and satisfy *cond*. When *init* is an initial state of $\mathcal{S}$ and the negation of a state predicate concerned is expressed in *pattern* and *cond*, the search functionality conducts BMC of an invariant, namely that it exhaustively traverses $\mathcal{R}^{\leq d}_{\mathcal{S},init}$ to find a counterexample showing that the state predicate is not an invariant.

$\mathcal{S}_{\mathrm{FMP}}$ is used as an example to describe how to specify transitions in transition rules. To specify transitions in transition rules, it is necessary to design the configuration of states. Associative-commutative collections of values observed by observers can be used as the configuration. For the configuration, the following are declared: "`op void : -> Sys {constr}`" and "`op __: Sys Sys -> Sys {constr assoc comm id: void}`". `Sys` is the sort denoting states, which are constructed with `void` and the juxtaposition operator. The juxtaposition operator is associative, commutative, and has `void` as its identity.

Since $\mathcal{S}_{\mathrm{FMP}}$ has two observers, the following two operators that hold two kinds of values observed by the two observers are declared: "`op (pc[_]:_) : Pid Label -> Obs {constr}`" and "`op locked:_: Bool -> Obs {constr}`". `Obs` is a subsort of `Sys`. Therefore, a collection of terms whose sorts are `Obs` denotes a state.

If two processes `p1` and `p2` participate in the protocol, the initial state (denoted by `init`) is expressed as follows: "`eq init = (pc[p1]: rs) (pc[p2]: rs) (locked: false) .`".

Let S, I, J, L1, L2 and B be CafeOBJ variables of sorts `Sys`, `Pid`, `Pid`, `Label`, `Label` and `Bool`, respectively, in the rest of the section. The three transitions are specified in transition rules as follows:

```
trans [try] : (pc[I]: rs) (locked: false)
  => (pc[I]: es) (locked: false) .
trans [enter] : (pc[I]: es) (locked: B)
  => (pc[I]: cs) (locked: true) .
trans [exit] : (pc[I]: cs) (locked: B)
  => (pc[I]: rs) (locked: false) .
```

where `try`, `enter` and `exit` enclosed with "`[`" and "`]`" are the labels (names) of the three transition rules, respectively.

The following command (the search functionality) can be used to try to find a counterexample showing that $\mathcal{S}_{\mathrm{FMP}}$ does not enjoy the mutual exclusion property: "`red init =(1,*)=>* (pc[I]: L1) (pc[J]: L2) S suchThat (not (L1 == cs and L2 == cs implies I == J)) .`". The command can be equivalently transformed into "`red init =(1,*)=>* (pc[I]: cs) (pc[J]: cs) S .`". Each of the commands can find a counterexample showing that $\mathcal{S}_{\mathrm{FMP}}$ does not enjoy the mutual exclusion property.

Since a state in which `inv1` does not hold is located at depth 4 from the initial state, the following command does not find the counterexample: "`red init =(1,3)=>* (pc[I]: cs) (pc[J]: cs) S .`".

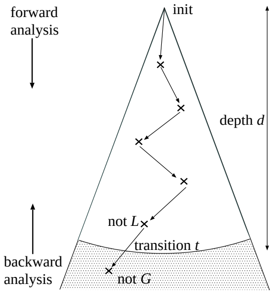

Fig. 1. Forward & backward reachability analysis

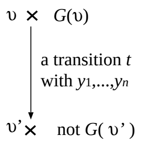
υ × $G(\upsilon)$
a transition t with $y_1,\ldots,y_n$
υ'× not $G(\upsilon')$

Fig. 2. A situation requesting a lemma in induction

4 Forward & Backward Reachability Analysis

Our primary goal is to discover a counterexample showing that a state predicate is not an invariant wrt an OTS $\mathcal{S}$.

4.1 Forward Reachability Analysis

Forward reachability analysis is to start with initial states and traverse the reachable state space to find some states in which some conditions hold (see Figure 1). Model checking, especially BMC, is a typical forward reachability analysis method. The search functionality is also a forward reachability analysis method. The method is fascinating as well as powerful in that it can fully automatically discover a counterexample showing that a state predicate is not an invariant. This is how we have found a counterexample showing that $\mathcal{S}_{\mathrm{FMP}}$ does not enjoy the mutual exclusion property in §3.

4.2 Backward Reachability Analysis

Backward reachability analysis method is to start with some states $\upsilon_1, \ldots, \upsilon_n$ (which may or may not be reachable) such that a state predicate does not hold and traverse the state space backward-reachable from $\upsilon_1, \ldots, \upsilon_n$ to check if an initial state of $\mathcal{S}$ is backward-reachable from υ_i for some $i \in \{1, \ldots, n\}$ (see Figure 1). If an initial state of $\mathcal{S}$ is backward-reachable from υ_i, then υ_i is reachable and then the predicate is not an invariant. If any initial state is not backward-reachable from an arbitrary state in which the predicate does not hold, we can conclude that the predicate is an invariant.

Structural induction can be regarded as a backward reachability analysis method. Let us consider an induction case for a transition t, together with $y_1, \ldots, y_n$. Let υ' be $t(\upsilon, y_1, \ldots, y_n)$ for an arbitrary state υ and G be a state predicate concerned. If $G(v)$ and $\neg G(v')$ (see Figure 2), then all we are concerned with is whether v is reachable. If it is, G is not an invariant. Otherwise, this induction case is discharged. To this end, what we can do is to conjecture a lemma. Although we do not know $\mathrm{depth}_{\mathcal{S}}(\mathit{init}, v)$ for some initial state *init* nor whether v is reachable, it is true that v is backward-reachable from v'. That is to say, one step is taken back from a state such that G does not hold by structural induction. This is how we have found a counterexample showing that $\mathcal{S}_{\mathrm{FMP}}$ does not enjoy the mutual exclusion property in §2.2.

4.3 Combination

Both forward and backward reachability analysis methods have the pros and cons. The search functionality can fully automatically discover a counterexample showing that a state predicate is not an invariant. This is, however, only the case when a state in which the predicate does not hold is located at a position that is not far from a given initial state *init*. The distance to a state v from *init* is not simply $\mathrm{depth}_{\mathcal{S}}(\mathit{init}, v)$. Let d be $\mathrm{depth}_{\mathcal{S}}(\mathit{init}, v)$ and then the distance crucially depends on the number of states in $\mathcal{R}^{\leq d}_{\mathcal{S},\mathit{init}}$.

If the reachable state space is huge or unbounded, there exists an upper bound d such that $\mathcal{R}^{\leq d}_{\mathcal{S},\mathit{init}}$ can be exhaustively traversed but $\mathcal{R}^{\leq d+1}_{\mathcal{S},\mathit{init}}$ cannot. If that is the case, the search functionality may not discover any counterexamples even though there exist some (see Figure 1). This is due to the notorious state explosion problem. If $\mathcal{R}^{\leq 4}_{\mathcal{S}_{\mathrm{FMP}},\mathtt{init}}$ was too large, the search functionality would not find a counterexample showing that the protocol does not enjoy the mutual exclusion property.

As described in §2.2, structural induction can be used to find a counterexample showing that a state predicate is not an invariant wrt an OTS $\mathcal{S}$. Generally, however, we need to conjecture a lot of necessary lemmas to have one such that it does not hold for some initial states. It also costs more than the search functionality.

But, structural induction may alleviate the state explosion problem, which bothers the search functionality. If $\mathcal{R}^{\leq 4}_{\mathcal{S}_{\mathrm{FMP}},\mathtt{init}}$ was too large, we could try to find a counterexample for `inv2`, which is a necessary lemma of `inv1`. The following command finds a counterexample: "`red init =(1,3)=>* (pc[I]: es) (pc[J]: cs) S .`". The command lets us know the reachable state `(pc[p2]: es) (pc[p1]: cs) (locked: true)` in which `inv2` does not hold. Consequently, `inv1` is not an invariant wrt $\mathcal{S}_{\mathrm{FMP}}$, either.

This is how we have come up with one possible way to complement each other, which is called induction-guided falsification (IGF)[4]. IGF is a combination of the search functionality (or BMC) and structural induction, but can be regarded as a combination of forward and backward reachability analysis methods because structural induction can be regarded as a backward reachability analysis method. If we exactly obey IGF, namely that every lemma conjectured is a necessary lemma, then once you find a counterexample for a lemma, you can quickly conclude that the original state predicate is not invariant. Even if non-necessary lemmas are used, the basic concept behind IGF, namely a combination of forward and backward reachability analysis methods, can be used. Non-necessary lemmas are less complicated than necessary lemmas.

Given an OTS $\mathcal{S}$ and a state predicate p, let $\mathcal{L}_{\mathcal{S},p}$ be a set of lemmas that can discharge the proof score that p is an invariant wrt $\mathcal{S}$. A variant of IGF is as follows:

```
Input: an OTS $\mathcal{S}$, a state predicate $p$, a natural number $d$;
Output: Verified or Falsified that $p$ is an invariant wrt $\mathcal{S}$;
1. $\mathcal{P} :=$ enqueue(empty-queue, $p$) and $\mathcal{Q} := \emptyset$.
2. Repeat the following until $\mathcal{P} =$ empty-queue.
3.     $q :=$ top($\mathcal{P}$) and $\mathcal{P} :=$ dequeue($\mathcal{P}$).
4.     Search $R_{\mathcal{S}}^{\leq d}$ for a state $\upsilon$ such that $\neg q(\upsilon)$.
       If such a state is not found, go to 8.
5.     Search $R_{\mathcal{S},\upsilon}^{\leq d}$ for a state such that $\neg p(\upsilon)$.
       If such a state is found, terminate and return Falsified.
6.     Search $R_{\mathcal{S},\upsilon}^{\leq d}$ for a state $\upsilon$ such that $\neg \mathrm{main}_q(\upsilon)$, where $\mathrm{main}_q$ is a state
       predicate in $\mathcal{Q}$, for which $q$ is used as a lemma.
       If such a state is found, $q := \mathrm{main}_q$, delete $q$ and the state predicates
       that are used as lemmas only for $q$ from $\mathcal{P}$ and $\mathcal{Q}$ and go to 5.
7.     Find a lemma $q'$ of $\mathrm{main}_q$ such that $q \Rightarrow q'$ and $q'$ is not equivalent to $q$,
       $q := q'$ and go to 4.
8.     Compute $\mathcal{L}_{\mathcal{S},q}$ by structural induction on $\mathcal{R}_{\mathcal{S}}$.
9.     $\mathcal{Q} := \mathcal{Q} \cup \{q\}$ and enqueue each in $\mathcal{L}_{\mathcal{S},q} - (\mathcal{Q} \cup \mathrm{q2s}(\mathcal{P}))$ into $\mathcal{P}$, where
       q2s($\mathcal{P}$) is the set that consists of the elements of $\mathcal{P}$.
10. Terminate and return Verified.
```

For example, if `inv2` was not a necessary lemma of `inv1`, the following command would find a counterexample for `inv1`: "`red (pc[p2]: es) (pc[p1]: cs) (locked: true) =(1,3)=>* (pc[I]: cs) (pc[J]: cs) S .`".

5 Application of the Variant of IGF to NSPK

5.1 NSPK and Agreement Property

NSPK[7] can be described as the three message exchanges:

Init: $p \rightarrow q \quad \{n_p, p\}_{k(q)}$
Resp: $q \rightarrow p \quad \{n_p, n_q\}_{k(p)}$
Ack: $p \rightarrow q \quad \{n_q\}_{k(q)}$

Each principal such as p and q is given a pair of keys (public and private keys). $\{m\}_{k(x)}$ is the ciphertext obtained by encrypting m with the principal x's public key. n_x is a nonce generated by a principal x.

The agreement property is as follows. Whenever a protocol run has been successfully completed by p and q,

AP1 the principal that p is communicating with is really q, and
AP2 the principal that q is communicating with is really p.

5.2 Specification for Structural Induction

We use the standard assumptions for protocol verification. Among them are that the cryptsystem used is perfect and the behaviors of malicious principals are formalized by the Dolev-Yao intruder[12]. Since we are only interested in invariants, it is not necessary to consider blocking of messages by the intruder.

A nonce generated by p for sending it to q is denoted by a term $\mathtt{n}(p, q, r)$ whose sort is `Nonce`, where r is a random number making the nonce unguessable and unique. Our formalism of NSPK allows a principal to participate in multiple sessions simultaneously. For each session, a principal needs to generate a fresh nonce.

Ciphertexts $\{n_p, p\}_{k(q)}$, $\{n_p, n_q\}_{k(p)}$ and $\{n_q\}_{k(q)}$ used in Init, Resp and Ack messages, respectively, are denoted by terms $\mathtt{enc1}(q, n_p, p)$, $\mathtt{enc2}(p, n_p, n_q)$ and $\mathtt{enc3}(q, n_q)$, respectively. Their sorts are $\mathtt{Cipher}i$ for $i = \mathtt{1}, \mathtt{2}, \mathtt{3}$, respectively.

Init, Resp and Ack messages are denoted by terms $\mathtt{m}i(p?, p, q, e_i)$ for $i = \mathtt{1}, \mathtt{2}, \mathtt{3}$, respectively. Their sorts are $\mathtt{Message}i$ for $i = \mathtt{1}, \mathtt{2}, \mathtt{3}$, respectively. The first argument $p?$ is a creator (an actual sender) of the message, the second argument p a seeming sender, the third argument q a receiver and the fourth argument e_i a ciphertext. The first argument is meta-information in that when q receives $\mathtt{m}i(p?, p, q, e_i)$, q cannot loot at $p?$. If $p?$ is different from p, then $p?$ is the intruder and the message has been faked by the intruder.

The network is formalized as an associative-commutative collection of messages whose sort is `Network`. Associative-commutative collections may be called just collections. A constant `empty` and a juxtaposition operator are the constructors of collections of not only messages but also the others such as nonces. We suppose that once a message $\mathtt{m}i(p?, p, q, e_i)$ is put into the network, it will be never deleted, and if there exists such a message in the network, q can receive it. When q has received it, q thinks that it originates in p.

We formalize the behaviors of NSPK as an OTS $\mathcal{S}_{\mathrm{NSPK}}$. We use three observers that are denoted by the observation operators: "`op network : System -> Network`", "`op rands : System -> RandSoup`", and "`op nonces : System -> NonceSoup`", where `System` is a sort denoting the (reachable) state space, `RandSoup` a sort denoting collections of random numbers, and `NonceSoup` a sort denoting collections of nonces. Given a state s, $\mathtt{network}(s)$ returns the network, the collections of messages that haven been sent up to s, $\mathtt{rands}(s)$ the collection of (old) random numbers that have been used up to s, and $\mathtt{nonces}(s)$ the collection of nonces that have been gleaned by the intruder up to s.

An arbitrary initial state is denoted by a constant `init` whose sort is `System`. `init` is a constructor of `System`. The three observation operators return `empty` for `init`.

Three transitions are used to formalize sending Init, Resp and Ack messages exactly obeying the protocol, respectively. The corresponding transition operators are as follows: "`op sdm1 : System Principal Principal Random -> System {constr}`", "`op sdm2 : System Principal Principal Principal Random Nonce -> System {constr}`", and "`op sdm3 : System Principal Principal Principal Nonce Nonce -> System {constr}`".

The set of equations defining sdm2 is as follows:

```
ceq network(sdm2(S,Q?,P,Q,R,N)) = m2(P,P,Q,enc2(Q,N,n(P,Q,R)))
 network(S)  if c-sdm2(S,Q?,P,Q,R,N) .
ceq rands(sdm2(S,Q?,P,Q,R,N)) = R rands(S) if c-sdm1(S,P,Q,R) .
ceq nonces(sdm2(S,Q?,P,Q,R,N)) = (if Q = intruder then N n(P,Q,R)
 nonces(S) else nonces(S) fi)  if c-sdm2(S,Q?,P,Q,R,N) .
ceq sdm2(S,Q?,P,Q,R,N) = S if not c-sdm2(S,Q?,P,Q,R,N) .
```

where c-sdm2(S,Q?,P,Q,R,N) is m1(Q?,Q,P,enc1(P,N,Q)) \in network(S) and not(R \in rands(S)).

The remaining two transition operators can be defined likewise. Symbols that appear in terms and are composed of capitals, numerals and ? are CafeOBJ variables in this section. Among them are S, Q? and RS2. Theirs sorts can be understood from the context.

c-sdm2(S,Q?,P,Q,R,N) says that there exists an Init message that seems to have been sent to Q by P in the network and R is a fresh random number. If that is the case, Q can receive the message and finds that the message obeys the protocol. Then, the Resp message m2(P,P,Q,...) is put into the network as the reply to the Init message. Since R is used in m2(P,P,Q,...), it is put into the collection of old random numbers. If Q is the intruder, the intruder can decrypt the ciphertext in m2(P,P,Q,...) and obtain the two nonces in it, which are put into the collection of nonces. Otherwise, the collection of nonces does not change. If c-sdm2(S,Q?,P,Q,R,N) does not hold. nothing changes. Receiving messages are implicitly formalized in transition operators.

Two kinds of values can be used to fake messages: messages and nonces. Since there are three kinds of messages, we use six transitions to formalize faking messages based on the gleaned information by the intruder. Due to the space limitation, we only describe two transitions faking Resp messages based on messages and nonces, respectively. The corresponding transition operators are as follows: "op fkm21 : System Principal Principal Message2 -> System {constr}" and "op fkm22 : System Principal Principal Nonce Nonce -> System {constr}.

The remaining four transition operators can be declared likewise.

The set of equations defining fkm22 is as follows:

```
ceq network(fkm22(S,P,Q,N1,N2))  = m2(intruder,P,Q,enc2(Q,N1,N2))
   network(S)  if c-fkm22(S,P,Q,N1,N2) .
eq rands(fkm22(S,P,Q,N1,N2)) = rands(S) .
eq nonces(fkm22(S,P,Q,N1,N2)) = nonces(S) .
ceq fkm22(S,P,Q,N1,N2) = S if not c-fkm22(S,P,Q,N1,N2) .
```

where c-fkm22(S,P,Q,N1,N2) is N1 \in nonces(S) and N2 \in nonces(S) and not(N1 = N2). c-fkm22(S,P,Q,N1,N2) says that the intruder has gleaned two different nonces. If that is the case, the intruder can fake a Resp message m2(intruder,P,Q,...), which is put into the network. Otherwise, nothing changes.

fkm21 and the remaining four transition operators can be defined likewise.

5.3 Specification for Search

Since there are the three observers, the following three operators are used to hold the values observed by them: "`op network:_: Network -> Obs {constr}`", "`op rands:_: RandSoup -> Obs {constr}`", and "`op nonces:_: NonceSoup -> Obs {constr}`". In addition to them, two more operators are used to hold two values: "`op prins:_: PrinSoup -> Obs {constr}`" and "`op rands2:_: RandSoup -> Obs {constr}`", where `PrinSoup` is a sort denoting collections of principals. The first operator holds a collection of principals participating in the protocol and the second a collection of fresh random numbers that can be used in the protocol. The two values are not modified by any transitions.

We suppose that three principals including the intruder participate in the protocol and two fresh random numbers are available. Then, the initial state denoted by `init` is expressed as "`(network: empty) (rands: empty) (nonces: empty) (prins: (p q intruder)) (rands2: (r1 r2))`".

The transition denoted by `sdm2` is specified in the following transition rule:

```
ctrans [sdm2] : (network: (m1(Q?,Q,P,enc1(P,N,Q)) NW))
 (rands: RS) (nonces: NS) (rands2: (R RS2))
 => (network: (m2(P,P,Q,enc2(Q,N,n(P,Q,R)))
              m1(Q?,Q,P,enc1(P,N,Q)) NW))
 (rands: (R RS))
 (nonces: (if Q == intruder then N n(P,Q,R) NS else NS fi))
 (rands2: (R RS2))  if not(R \in RS) .
```

The transition denoted by `fkm22` is specified in the following transition rule:

```
trans [fkm22] :
 (network: NW) (nonces: (N1 N2 NS)) (prins: (P Q PS))
 => (network: (m2(intruder,P,Q,enc2(Q,N1,N2)) NW))
 (nonces: (N1 N2 NS)) (prins: (P Q PS)) .
```

The remaining seven transitions can be specified likewise.

5.4 Falsification

AP2 is formalized in terms of the following state predicate `inv2`:

```
eq inv2(S,P,Q,P?,R,N) = (not(Q = intruder) and
 m2(Q,Q,P,enc2(P,N,n(Q,P,R))) \in network(S) and
 m3(P?,P,Q,enc3(Q,n(Q,P,R))) \in network(S)
 implies m3(P,P,Q,enc3(Q,n(Q,P,R))) \in network(S)) .
```

AP1 can also be formalized likewise.

What we did first is to find an upper bound d such that $\mathcal{R}^{\leq d}_{\mathcal{S}_{\mathrm{NSPK}},\mathtt{init}}$ can be exhaustively traversed as follows: "`red init =(1,5)=>* S suchThat false .`". On a laptop with 2.33GH CPU and 3GB RAM, 5 was the upper bound[2].

[2] Since the implementation of the CafeOBJ search functionality was not matured enough, Maude was used to conduct the experiment described in this section.

In $\mathcal{R}^{\leq 5}_{\mathcal{S}_{\mathrm{NSPK}},\mathtt{init}}$, no counterexample was discovered for inv1 and inv2. The following command tries to find a counterexample for inv2:

```
red init =(1,5)=>* (network: (m2(Q,Q,P,enc2(P,N,n(Q,P,R)))
 m3(P?,P,Q,enc3(Q,n(Q,P,R))) NW)) S
 suchThat (not(not(Q == intruder) implies
 m3(P,P,Q,enc3(Q,n(Q,P,R))) \in m3(P?,P,Q,enc3(Q,n(Q,P,R))) NW)) .
```

Then, we conjectured lemmas to discharge the proof scores of inv1 and inv2. Five lemmas were conjectured. Two out of them are as follows:

```
eq inv4(S,P,Q,N,R,M2) = (not(P = intruder) and not(Q = intruder)
  and m1(P,P,Q,enc1(Q,n(P,Q,R),P)) \in network(S) and
  M2 \in network(S) and cipher2(M2) = enc2(P,n(P,Q,R),N)
  implies m2(Q,Q,P,enc2(P,n(P,Q,R),N)) \in network(S)) .
eq inv5(S,N) = (N \in nonces(S)
  implies creator(N) = intruder or forwhom(N) = intruder) .
```

Each of the five lemmas is a necessary one of neither inv1 not inv2.

No counterexample was found in $\mathcal{R}^{\leq 5}_{\mathcal{S}_{\mathrm{NSPK}},\mathtt{init}}$ for the four lemmas including inv4. But, a counterexample was found for inv5, which formalizes what is called the nonce secrecy property. Hence, NSPK does not enjoy the nonce secrecy property. Since inv5 is a necessary lemma of neither inv1 nor inv2, however, we cannot immediately conclude that NSPK does not enjoy the agreement property.

The state in which inv5 does not hold is as follows:

```
eq s115890 = (nonces: (n(q,p,r2) n(p,intruder,r1)))
  (network: ( m1(intruder,p,q,enc1(q,n(p,intruder,r1),p))
   m1(p,p,intruder,enc1(intruder,n(p,intruder,r1),p))
   m2(intruder,intruder,p,enc2(p,n(p,intruder,r1),n(q,p,r2)))
   m2(q,q,p,enc2(p,n(p,intruder,r1),n(q,p,r2)))
   m3(p,p,intruder,enc3(intruder,n(q,p,r2)))))
  (rands: (r1 r2)) (prins: (intruder p q)) (rands2: (r1 r2)) .
```

The state, which is reachable, is reported by the search functionality. This is the 115890th state that the search functionality has visited from init.

Instead of $\mathcal{R}^{\leq 5}_{\mathcal{S}_{\mathrm{NSPK}},\mathtt{init}}$, we can then traverse $\mathcal{R}^{\leq 5}_{\mathcal{S}_{\mathrm{NSPK}},\mathtt{s115890}}$ to find a counterexample for inv1 and inv2. But, $\mathcal{R}^{\leq 5}_{\mathcal{S}_{\mathrm{NSPK}},\mathtt{s115890}}$ was too large to be exhaustively traversed. Therefore, we traversed $\mathcal{R}^{\leq 4}_{\mathcal{S}_{\mathrm{NSPK}},\mathtt{s115890}}$ to find a counterexample for inv1 and inv2. The command to find a counterexample for inv2 is as follows:

```
red s115890 =(1,4)=>* (network: (m2(Q,Q,P,enc2(P,N,n(Q,P,R)))
 m3(P?,P,Q,enc3(Q,n(Q,P,R))) NW)) S
 suchThat (not(not(Q == intruder) implies
 m3(P,P,Q,enc3(Q,n(Q,P,R))) \in m3(P?,P,Q,enc3(Q,n(Q,P,R))) NW)) .
```

No counterexample was found for inv1 but a counterexample was found for inv2. The counterexample found is the same as that was found by Lowe[13].

6 Related Work

Another possible combination of BMC and structural induction has been proposed: k-induction[14]. It has been implemented in SAL[15], which is a toolkit for analyzing state machines. The primary purpose of k-induction is verification. k-induction can be used to verify that a system (formalized as a state machine) enjoys an invariant. It is, however, necessary to fix the number of entities (such as processes) participating in a system. Since standard structural induction is used in (the variant of) IGF, only one step is taken back from a state in which a state predicate concerned does not hold. k-induction allows to take more than one step back from such a state. Hence, it may make (the variant of) IGF more powerful to adopt k-induction, which is one piece of our future work.

Maude-NPA[3] has been implemented in Maude[6], relying on the narrowing search functionality. While the term *init* should be ground in the ordinary search functionality, it can contain variables in the narrowing search functionality. Hence, *init* can express an arbitrary state in which a state predicate concerned does not hold. The (ordinary and narrowing) search functionality can conduct a backward reachability analysis by reversing the transition rule specifying each transition. This is how Maude-NPA conducts a backward reachability analysis. The backward reachability analysis method used by Maude-NPA may be used for (the variant of) IGF. If so, we only need to have one type of specifications in which transitions are specified in transition rules. This is another piece of our future work. The narrowing search functionality may be used to implement more general k-induction such that it is not necessary to fix number of entities participating in a system. This is yet another piece of our future work.

7 Conclusion

The primary purpose of (the variant of) IGF is to falsify that a system enjoys a property. The mainly used technique for this purpose is testing, which can be roughly classified into exhaustive and non-exhaustive testing. (Bounded) Model checking can be used for the former. Daniel Jackson, who is the main designer of Alloy[2], has formed the small scope hypothesis, which says that most errors can be found by testing a program for all test inputs within some small scope[16]. This implies that it is more beneficial to exhaustively test a program within some small scope than to test it for some randomly generated test cases within larger scope. This is why Alloy has adopted a SAT-based bounded model checker.

The state in which AP2 (`inv2`) does not hold has not been found within the small scope such that the search functionality can exhaustively traverse the scope. Some may suggest that the nonce secrecy property should be taken into account instead of the agreement property because the former is more fundamental than the latter for authentication protocols. This is why almost all analyzes of NSPK have taken into account the nonce secrecy property[8, 9, 3]. Generally, however, we do not know in advance what is more fundamental than a property concerned such as the agreement property for a system such as NSPK. Therefore, we need to extend the scope that can be exhaustively traversed so as to find more errors. This is why (the variant of) IGF has been proposed and a backup case study has been conducted.

One piece of our future work is to design and implement a tool supporting (the variant of) IGF. We may use the translator[17] from state machine specifications in CafeOBJ into those in Maude and the technique to discover lemmas used in Crème[18], an automatic invariant prover for state machine specifications in CafeOBJ.

References

1. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic model checking without BDDs. In: 5th TACAS. LNCS 1579, Springer (1999) 193–207
2. Jackson, D.: Alloy: A lightweight object modeling notation. ACM TOSEM **11** (2002) 256–290
3. Escobar, S., Meadows, C., Meseguer, J.: A rewriting-based inference system for the NRL protocol analyser and its meta-logical properties. TCS **367** (2006) 162–202
4. Ogata, K., Nakano, M., Kong, W., Futatsugi, K.: Induction-guided falsification. In: 8th ICFEM. LNCS 4260, Springer (2006) 114–131
5. Diaconescu, R., Futatsugi, K.: CafeOBJ report. AMAST Series in Computing, 6. World Scientific (1998)
6. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All About Maude – A High-Performance Logical Framework: How to Specify, Program and Verify Systems in Rewriting Logic. LNCS 4350. Springer (2007)
7. Needham, R.M., Schroeder, M.D.: Using encryption for authentication in large networks of computers. CACM **21** (1978) 993–999
8. Lowe, G.: Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In: 2nd TACAS. LNCS 1055, Springer (1996) 147–166
9. Denker, G., Meseguer, J., Talcott, C.: Protocol specification and analysis in Maude. In: Workshop on Formal Methods and Security Protocols. (1998)
10. Ogata, K., Futatsugi, K.: Some tips on writing proof scores in the OTS/CafeOBJ method. In: Algebra, Meaning, and Computation: A Festschrift Symposium in Honor of Joseph Goguen. LNCS 4060, Springer (2006) 596–615
11. Gâinâ, D., Futatsugi, K., Ogata, K.: Constructor-based institutions. In: 3rd CALCO. LNCS 5728, Springer (2009) 398–412
12. Dolev, D., Yao, A.C.: On the security of public key protocols. IEEE TIT **IT-29** (1983) 198–208
13. Lowe, G.: An attack on the Needham-Schroeder public-key authentication protocol. IPL **56** (1995) 131–133
14. de Moura, L., Rueß, H., Sorea, M.: Bounded model checking and induction: From refutation to verification. In: 15th CAV. LNCS 2392, Springer (2003) 14–26
15. de Moura, L., Owre, S., Rueß, H., Rushby, J., Shankar, N., Sorea, M., Tiwari, A.: SAL 2. In: 16th CAV. LNCS 3114, Springer (2004) 496–500
16. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. MIT Press (2006)
17. Zhang, M., Ogata, K., Nakamura, M.: Specification translation of state machines from equational theories into rewrite theories. In: 12th ICFEM. LNCS (this volume), Springer (2010)
18. Nakano, M., Ogata, K., Nakamura, M., Futatsugi, K.: Crème: An automatic invariant prover of behavioral specifications. IJSEKE **17** (2007) 783–804