Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science 254 (2009) 105–122
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Modular Checking with Model Checking

Yuusuke Hashimoto[1]

The Graduate University for Advanced Studies
and
NEC Corporation
Tokyo, Japan

Shin Nakajima[2]

National Institute of Informatics
and
The Graduate University for Advanced Studies
Tokyo, Japan

Abstract

Automatic static checkers based on model checking, particularly SAT-based bounded model checkers, are used in industry, but they sometimes suffer from the scalability problem. Scalability can be achieved with the notions of Design by Contract(DbC) and modular checking. However, modular checking with DbC still have some problems. The method is insufficient for handling pointers to functions (function-pointers) which are abundantly used in C programs, defensive programming which is widely adopted in industrial software development projects, and re-entrancy which sometimes occurs in programs using callback functions. This paper proposes a DbC notation for the above problems and a checking method that uses behavioral subtyping to clarify the exact location where an error occurs.

Keywords: C program, model checking, modular checking, pointer to function, re-entrancy

1 Introduction

Model checking of C and Java programs has been extensively studied, and some industrial strength tools are now available [1,2,12]. In particular, SAT-based bounded model checkers [4,8,13] can find subtle bugs. However, they

[1] Email: yu-hash@nii.ac.jp or yu-hash@cb.jp.nec.com
[2] Email: nkjm@nii.ac.jp

doi:10.1016/j.entcs.2009.09.062

suffer from the state space explosion problem, which results in them having poor scalability. Users sometimes have to pre-process the program codes to cut down its size before they begin checking with the tool. The resulting witnesses of errors also need to be manually checked afterwards. A systematic means of modular checking is desirable to overcome this problem.

Modular checking has been realized in the form of extended static checkers [9,10]. These checkers use the notion of Design by Contract(DbC) [16] and separate the program check into caller side and callee side parts. The behavioral specifications of the callee function or procedure (called contracts) are the key to making the separation. The caller side check uses the externally visible behavior of the callee, and the callee's procedure body can be checked regardless of its caller; the check is done solely using the behavioral specifications. Since the checked program is small, the scalability problem disappears, but at the risk of generating spurious alarms. Furthermore, the re-entrancy problem [7] manifests itself for certain programs because the analysis cannot make use of the caller side information.

VARVEL, a SAT-based bounded model checker for C programs, combines model checking and modular checking. It performs modular checking on top of the model checker F-Soft [13]. The idea of introducing DbC to a program model checker is not new. For example, CBMC [4] and Bogor [18] as well as F-Soft provide two primitives, *assume* and *assert*. These primitives can be combined to emulate behavioral specifications needed for modular checking. However, VARVEL explicitly introduces such a DbC style, which means that the tool automatically embeds the appropriate primitives corresponding to a contract at specific positions in the source programs and conducts model checking.

Furthermore, VARVEL provides advanced features to deal with DbC for higher-order functions [11]. In C programming, higher-order functions are emulated by using function-pointers. The capability of handling function-pointers is mandatory for checking industrial C programs because they use many function pointers.

In this paper, we illustrate how the model checker we developed (VARVEL) can provide modular checking, i.e., modular checking with model checking. We also discuss DbC for function-pointers and the case of defensive programming. The code snippets in the paper are from source programs developed in industry that VARVEL has been applied to.

2 Background

Model checking constructs a finite-state automaton from the specifications of the target system to be checked and exhaustively searches the state space to collect the states satisfying a given property [3]. In general, model checking suffers from the state space explosion problem.

Application of model checking to programs (modern model checking) has further problems, such as handling data values, procedure calls and so on. Finite-state automaton does not explicitly represent the procedure calls, and many modern model checkers perform the in-line expansion of a sequence of procedures. This makes the state space explosion problem worse and results in poor scalability. Especially, performing the flow-sensitive analysis of a program additionally requires caller side programs, and the size of programs to be analyzed at a time becomes huge.

Client program

```
  x = ... ;
  ... ;
  y = deposit( x ) ;
  ... ;
```

Procedure

```
unsigned int  bal ;
/**
    @invariant  0 <= bal && bal < MAX
*/

/**
    @pre   0 < x
    @post  __return == bal && bal == __old( bal ) + x
    @param[out]   bal
*/
int  deposit( unsigned int  x ){
    int  result;
    if( bal + x < MAX )
        result = ( bal += x ) ;  /* Success */
    else
        result = -1 ;            /* Error */
    return  result ;
}
```

Fig. 1. Example of DbC Description

Extended static checkers [9,10] enable modular checking through the notion of DbC [16]. These checkers divide the program check into caller side and callee side checks. In DbC, the callee program *Procedure* has a precondition Pre_P and a postcondition $Post_P$. In the example in Fig.1, the function `deposit` has a precondition (`@pre`), a postcondition (`@post`), and information about the variables to be modified (`@param[out]`) as its contract. In the postcondition, the symbol `__return` denotes the return value of the function, and the symbol `__old( expression )` denotes the value of the expression at the beginning of the function. In addition, the global variable `bal` has the invariant(`@invariant`). Programs are checked against contracts as follows. On the caller side, the calling program *ClientProgram* makes the execution state S satisfying Pre_P before calling *Procedure* without knowing how $Procedure$ is performed. $ClientProgram$ assumes that $Post_P$ satisfies the execution state R after returning from $Procedure$. This is denoted as

$$S \Rightarrow Pre_P, \quad Pre_P \wedge Post_P \Rightarrow R \qquad (1)$$

On the callee side, the callee program $Procedure$ executes the body of function $Body_P$ under the assumption of Pre_P and satisfies $Post_P$ as a result. This is denoted with the notion of the weakest precondition by E. Dijkstra [6] as

$$Pre_P \Rightarrow \text{wp}.Body_P.Post_P \qquad (2)$$

When the invariant is taken into account, it is denoted as

$$Pre_P \wedge Inv \Rightarrow \text{wp}.Body_P.(Post_P \wedge Inv') \qquad (2')$$

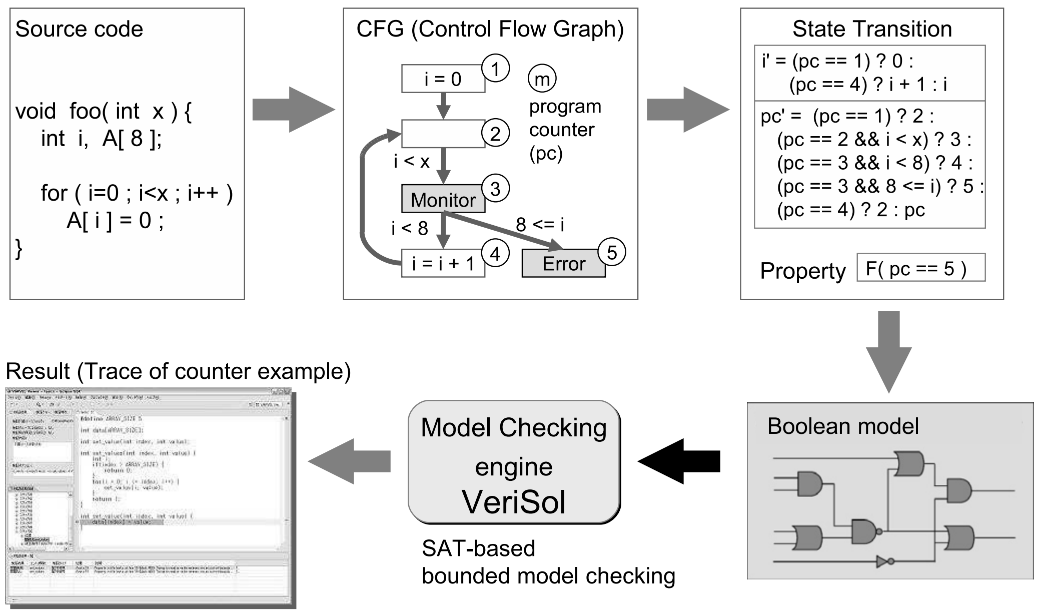
Inv' is the invariant after the function body was processed. Obviously from (1) and (2), it is possible to check the calling program and the called program separately; modular checking is performed and the method is scalable. A further advantage of DbC is that it clarifies where the responsibility lies for fixing a defect. For instance, if S does not satisfy Pre_P, the calling program $ClientProgram$ needs to be modified.

A disadvantage of modular checking is that it does not perform the inter-procedural flow-sensitive analysis because it stops the analysis at the boundary of the contracts and uses no caller side information. Therefore, modular checking produces many spurious witnesses of the errors. Particularly, it does not handle re-entrancy where a function of a module will be processed while the other function of the module is processed. In addition, the expressiveness of the contract notation has some problems, such that it does not handle higher order functions (function-pointers in the case of the C language). Also, DbC remains controversial to what should be denoted as contracts, especially for defensive programming.

The notion of DbC is independent of underlying checking technologies. Before the recent appearance of static checkers, programs were checked against contracts by testing [11,16]. SAT-based bounded model checkers provide some primitives (assertion, assumption, etc.) for describing user-defined properties and assumptions on the external environment of a checked program. DbC can be introduced to SAT-based bounded model checkers by using the checkers' primitives. VARVEL is based on the SAT-based bounded model checker F-Soft [13].

3 Overview of VARVEL

As mentioned above, VARVEL is a source codes checker for sequential C programs, and it is based on the SAT-based bounded model checker F-Soft



```
void  foo( int  x ) {
   int  i,  A[ 8 ];

   for ( i=0 ; i<x ; i++ )
      A[ i ] = 0 ;
}
```

```
i' = (pc == 1) ? 0 :
   (pc == 4) ? i + 1 : i
```

```
pc' =  (pc == 1) ? 2 :
   (pc == 2 && i < x) ? 3 :
   (pc == 3 && i < 8) ? 4 :
   (pc == 3 && 8 <= i) ? 5 :
   (pc == 4) ? 2 : pc
```

Fig. 2. Processing flow of VARVEL

[13]. Figure 2 shows the processing flow of VARVEL. It accepts ANSI-C (C90) compliant source codes as its input and performs the following procedures to detect typical errors and violations of user-defined assertions.

First, VARVEL transforms the input source codes into control flow graphs (CFGs). It simplifies the source codes, for instance, by combining multiple `return` statements of a function into a single `return` statement, converting a `for` statement into an equivalent `while` statement. This simplification compresses a variety of expressions from C programs into a subset of ANSI-C and makes it easy to parse the source codes and to construct CFGs. During the construction of CFGs, VARVEL adds three kinds of information. The first one is the implicit assumptions that supply the information on the external environment, such as the values of the global variables, the actual parameters of the *entry function* which is specified as the root of the call-tree, and the return value of the *library function* whose source codes are not available at the time of the analysis. The second one is the internal variables that are a program counter identifying the location on the CFGs and monitored variables indicating the bounds of an array, the validity of a pointer, etc. The third one is the internal nodes that are the monitoring nodes checking whether the monitored variables have proper values and the error nodes indicating the occurrences of errors. After the construction of the first CFGs, VARVEL performs a static analysis to decrease the size of the CFG. It removes the irrelevant nodes to the error nodes from the CFGs, and performs a points-to analysis to determine a set of memory addresses potentially assignable to a specific pointer variable. The results of the points-to analysis enables VARVEL to merge multiple CFGs into one CFG through the function-pointers.

Next, VARVEL generates state transition and property formulas from the CFG. The state transition is represented as a set of formulas. Each formula denotes the next value of a variable from the current values of some variables. A property is an LTL formula denoting the condition to reach the error nodes. The tool converts these formulas into boolean formulas and passes them along to the model checking engine, which then performs SAT-based bounded model checking on the given boolean formulas.

Finally, VARVEL translates the model checking results into the original source codes. A witness of the error obtained by the model checking is shown as a call stack from the beginning of the *entry function* to the location where an error occurs.

Providing the notion of DbC explicitly is unique to VARVEL. VARVEL uses F-Soft's primitives (assumption and assertion) to emulate DbC. F-Soft provides these primitives for users to describe the external environment information. The use of primitives by VARVEL varies according to the following functions.

- *Entry functions*
- *Library functions*
- Other functions: functions within a function call tree, except the *entry function* and the *library functions.*

Figure 3 shows the difference in usage of primitives. Case 0 is an example source codes and Case 1,2, and 3 are the usages of primitives for Case 0. `__assume(...)` is a primitive assumption and `__assert(...)` is a primitive assertion.

Case 1: Entry functions. For the *entry functions*, each precondition or invariant is converted into an assumption at the beginning of the function, and each postcondition or invariant is converted into an assertion at the end of the function.

VARVEL implicitly assumes that the variable such as the argument of the *entry function* and the global variable, which comes from outside of the functions to be checked, takes a special symbol referring to a non-deterministically chosen value. Assumptions that are specified as preconditions or invariants are applied after the implicit assumptions are applied.

Case 2: Library functions. The contract of a *library function* is associated with its prototype declaration and is given to VARVEL. For the *library functions*, each precondition or invariant is converted into an assertion at the

Case 0: Sample source codes

```
unsigned int g = 0;
/**
    @invariant  g <= 1
*/

/**
    @pre   0 < x
    @post  0 <= __return
    @param[out]  g
*/
int foo( int x ){
    int result ;

    /* function body */

    return  result;
}
```

Case 1: Entry functions

```
int foo( int x ){
    int result ;
    __assume( 0 < x ) ;
    __assume( g <= 1 ) ;

    /* function body */

    __assert( 0 <= result ) ;
    __assert( g <= 1 ) ;
    return  result;
}
```

Case 2: Library functions

```
int foo( int x ){
    int result ;
    __assert( 0 < x ) ;
    __assert( g <= 1 ) ;

    result = __NON_DET__();
    g = __NON_DET__();

    __assume( 0 <= result ) ;
    __assume( g <= 1 ) ;
    return  result;
}
```

Case 3: Other functions

```
int foo( int x ){
    int result ;
    __assert( 0 < x ) ;
    __assume( 0 < x ) ;
    __assert( g <= 1 ) ;
    __assume( g <= 1 ) ;

    /* function body */

    __assert( 0 <= result ) ;
    __assume( 0 <= result ) ;
    __assert( 0 <= __return ) ;
    __assume( 0 <= __return ) ;
    return  result;
}
```

Fig. 3. Example description of contracts

beginning of the function, and each postcondition or invariant is converted into an assumption at the end of the function. Non-deterministically chosen values are assigned to the variables to be modified before the postconditions are converted. In Fig.3, the global variable `g` is specified to be modified by declaration `@param[out] g` in Case 0. A return value is implicitly regarded as modified in the function.

Case 3: Other functions. For other functions, each precondition or invariant is converted into two primitives (assertion and assumption) at the beginning of the function, and each postcondition or invariant is also converted into two primitives at the end of the function.

4 Function-pointers

One of the problems we encountered in applying VARVEL to industrial programs is related to the function-pointers. In C programs, function-pointers are language constructs to realize higher-order functions. They are frequently used in C programs, such as device drivers, event dispatchers, GUI frameworks and so on.

In general, a higher-order function is one that takes a function as its argument or its return value. For instance, the following function `f` is a higher-order function that takes another function as its argument and returns an integer value and this other function takes an integer argument and returns an integer value.

$$f : (int \rightarrow int) \rightarrow int$$

Likewise, the following function `g` is a higher-order function that takes an integer argument and returns a function.

$$g : int \rightarrow (int \rightarrow int)$$

Functional languages that treat functions as first class data must be able to handle the contract of a higher-order function. There is an existing work that introduces contracts to Scheme [11], but the contracts are checked at run-time, as in Eiffel [16]. Our method differs from the existing work because it checks contracts statically.

A higher-order function is difficult to check against a contract statically, since the preconditions and postconditions of a callee function are not decided until the address of the callee function is assigned to a function-pointer. Modular checking divides a program into modules by regarding contracts as boundaries. This makes it even more difficult to use the source codes of the callee functions in other modules.

Figure 4 shows an example source codes using a function-pointer. The function `delegate` has a function-pointer `comp` as its formal argument, and the function `user` calls the `delegate` with the address of a function `comp` as the actual argument. `delegate`'s precondition `comp != NULL` indicates that the pointer argument `comp` dereferences a valid memory address. In this case, two actual arguments (`calc` and `-1`) of the `delegate` given by the `user` are consistent with the precondition of the `delegate`, so the check at the calling site of the `delegate` succeeds. A precondition of the `calc`, however, is violated at its calling site through the function-pointer `(*comp)(0)` in the `delegate`.

One way to fix this precondition violation is to explicitly define the precondition and postcondition of the formal argument `comp` of the `delegate` function. If the precondition is fixed as in Case 1 of Fig.4, the `delegate(calc, -1)` shows up as a defect. If the defect is fixed like `delegate(calc, 0)`, it satisfies the precondition of the `comp`, but it still violates the precondition of the `calc`. Defining only the contract of a function-pointer does not solve the problem. In order to check C programs with function-pointers, we need to solve the following problems.

Case 0:
Source code with function-pointer

```
/**
  @pre   comp != NULL && n >= -10
  @post  __result > 0
*/
int  delegate(
      int (*comp)(int), int n )
{
  int  x ;
  if ( n > 0 )
    x = (*comp)( n ) ;
  else
    x = (*comp)( 0 ) ;
  return  x ;
}

/**
  @pre   n > 0
  @post  __return > 0
*/
int  calc( int n ) { ... }

int  user( int m ) {
  return  delegate( calc, -1 );
}
```

Case 1:
Contracts for function-pointer

```
/**
  @pre   comp != NULL && n >= -10
  @post  __result > 0
  @pointer  comp
    @pre   __a1 >= 0
    @post  __return > 0
*/
int  delegate(
      int (*comp)(int), int n )
{    . . . . . .    }
```

Fig. 4. Example of Pointer to Function

- *formal contract* : How can we give a contract notation for a function-pointer (a *formal contract*) and check the source codes using the function-pointer against the *formal contract*.
- *actual contract* : How can we check the consistency of the contract of the function called via the function-pointer (the *actual contract*) with the *formal contract*.

4.1 Notation of Formal Contracts

We need to extend the notation of the contract to denote the contract associated with a function-pointer (a *formal contract*) for the following cases:

- when a (local or global) variable is a function-pointer,
- when a function argument is a function-pointer, and
- when a return value of a function is a function-pointer.

Some example notations for each of these cases are shown in Fig.5. The keyword `@pointer` *PointerName* denotes the start of a *formal contract*, and the subsequent preconditions and postconditions compose the *formal contract*. In the *formal contract*, the symbol `__aN` (N=1,2,...) represents the Nth argument of a function called via the function-pointer.

A *formal contract* should be checked when a function call via a function-pointer occurs. This can be achieved by inserting primitive assertions before

Case 1: For variables

```
void bar( )
{
  int  x, y ;
  int (*fp)( int ) ;
  /*
    @pointer  fp
      @pre  0 < __a1
      @post __return == 0
         || __return == 1
  */
  ...
  y = (*fp)( x ) ;
  ...
}
```

Case 2: For arguments

```
/**
  @pointer  fp
    @pre  0 < __a1
    @post __return == 0
       || __return == 1
*/
void bar(
    int (*fp)( int ) )
{
  int  x, y ;
  ...
  y = (*fp)( x ) ;
  ...
}
```

Case 3: For return values

```
/**
  @pointer  __return
    @pre  0 < __a1
    @post __return == 0
       || __return == 1
*/
int (*bar( void ))( int )
{
  int (*fp)( int ) ;
  ...
  fp = FUNCTION_ADDRESS ;
  ...
  return  fp ;
}
```

Fig. 5. Example of Notation for Formal Contract

the call and primitive assumptions after the call. Some examples are shown in Fig.6.

Case 1: For variables

```
void bar( )
{
  int  x, y ;
  int (*fp)( int ) ;
  ...
  __assert( 0 < __a1 ) ;
  y = (*fp)( x ) ;
  __assume( y == 0 || y == 1 ) ;
  ...
}
```

Case 2: For arguments

```
void bar( int (*fp)( int ) )
{
  int  x, y ;
  ...
  __assert( 0 < __a1 ) ;
  y = (*fp)( x ) ;
  __assume( y == 0 || y == 1 ) ;
  ...
}
```

Fig. 6. Assertions for Formal Contract

4.2 Consistency of Actual and Formal Contracts

The relationship between a function-pointer and dereferenced functions can be interpreted as substitutability. Behavioral subtyping denotes that the preconditions of the `subtype` satisfy the preconditions of the `supertype` and the postconditions of the `supertype` satisfy the postconditions of the `subtype` [15].

$$Pre_{sub} \Rightarrow Pre_{super}$$
$$Post_{sub} \Leftarrow Post_{super}$$

We propose the use of behavioral subtyping for the formal argument $comp_f$ and actual argument $comp_a$ of a function-pointer.

$$PreComp_f \Rightarrow PreComp_a$$
$$PostComp_f \Leftarrow PostComp_a$$

The consistency between a *formal contract* and an *actual contract* can be checked by using an automatic decision procedure or a theorem prover such as Simplify [5]. In the case in Fig.4, Simplify gives `x==0` as a counter-example of behavioral subtyping for the formulas in Fig.7. Predicates `comp_f_pre` and `comp_a_pre` respectively denote the preconditions of the formal argument $comp_f$ and the actual argument $comp_a$. The formula starting with $FORALL(x)$ checks the behavioral subtyping rule for the preconditions.

```
(DEFPRED (comp_f_pre a1) (<= 0 a1))
(DEFPRED (comp_a_pre n) (< 0 n))
(FORALL (x)
    (IMPLIES (comp_f_pre x) (comp_a_pre x))
)
```

Fig. 7. Formulas for Simplify

4.3 Dynamic Link Library

An explicit description of the contract for a local function-pointer is useful in some cases. Figure 8 is an example of a program calling a *library function* dynamically loaded and linked at run-time. The function `foo` assigns the address returned by a function `dlsym` to the function-pointer `comp` and calls the function `f` through the pointer. In order to check `foo` against its contract, it is necessary to describe the (preconditions and) postconditions of `comp`; otherwise, `(*comp)` returns the arbitrary value and the postcondition of `foo` is violated.

```
/**
  @post  0 <= __return
*/
int  foo( int  n ) {
  int  (*comp)( int ) ;
  /*
    @pointer  comp
      @pre  0 < __a1
      @post 0 < __return
  */

  int  x ;
  void  *h = dlopen( "x.so", RTLD_LAZY ) ;
  comp = ( int  (*)( int ) ) dlsym( h, "f" ) ;
  x = (*comp)( n ) ;
  dlclose( h ) ;
  return  x - 1 ;
}
```

Fig. 8. Dynamic Link Library

5 External and Internal Specifications

Another problem we sometimes encountered is what we specify as functions' contracts. We may give contracts in an adequate manner depending on how we use them. Contracts of the function define its specification. An external specification is determined from the functional design, while an internal specification is from the implementation policy regarding the non-functionality, such as the robustness.

Defensive programming is a technique for achieving robustness, and it has been used in many industrial software development projects. Programmers write as many checking codes for the invalid arguments as possible. From the viewpoint of the design or functionality, the contract of a function is an external specification of its intrinsic functionality to the other functions. In this sense, the contract of a function should not accept invalid arguments because they result in the function returning just an error code without performing its intrinsic functionality.

From the viewpoint of the implementation or robustness, the contract of a function is an internal specification describing all the behaviors of the function. In this sense, the contract of a function might accept any actual argument and the function should check its validity to defend the intrinsic processing from erroneous situations.

Case 0: Sample source codes

```
/* Returns absolute value of *p */
int  bar( int *p ){
    /* for robustness */
    if ( p == NULL ) return  -1;

    /* for intrinsic functionality */
    if ( 0 <= *p ) return  *p ;
    else  return -(*p) ;
}
```

Case 1: Contracts for functionality

```
@pre   p != NULL
@post  __return >=  0
```

Case 2: Contracts for robustness (defensive programming)

```
@post  p != NULL --> __return >=  0
    && p == NULL --> __return == -1
```

Case 3: Contracts for precise implementation

```
@pre   p != NULL
@post  0 <= __return
@warning
   @pre   p == NULL
   @post  __return == -1
```

Fig. 9. Example of Defensive Programming

Figure 9 shows an example of the defensive programming technique. In the example source codes (Case 0), a function `bar` checks the value of an argument `p` for robustness. If `p` is `NULL`, `bar` returns a `-1` (which is not an absolute value) to notify the callers that something wrong happened. Otherwise, `bar` calculates the absolute value of the variable dereferenced by pointer `p` and returns the calculated result. Case 1 is an example of a *external contract* that represents an external specification for the intrinsic functionality. The precondition `@pre` denotes that callers must not pass `NULL` to `bar` in order for `bar` to perform its intrinsic functionality. Case 2 is an example of a *internal contract* that represents an internal specification for defensive programming. No precondition indicates that `bar` accepts any actual argument, and the postcondition `@post` denotes the result for each case of an actual argument. If `p` is not `NULL`, `bar` returns a non-negative value. If `p` is `NULL`, `bar` returns a `-1`. Here, the symbol `-->` denotes logical *implies*. Functions calling `bar` must judge whether the intrinsic functionality has been performed or not from the return value of `bar`.

As for the callee side check, the contract in Case 2 of Fig.9 gives more precise information than that in Case 1, and is suitable for checking the function `bar` itself. As for the caller side check, the choice of whether to choose Case 1 or 2 depends on the programming policy. A software development project may adopt a policy to check the actual arguments of the functions before calling them and chooses the contract in Case 1 for static checkers. Another project may adopt a policy to check the return value of the functions without checking their actual arguments and chooses the contract in Case 2. Hence, it is preferable to describe both contract types so that programmers can choose the type of contract for static checkers.

Case 3 in Fig.9 is an idea for the notation to represent both contracts from Case 1 and 2 by distinguishing the intrinsic functionality and robustness. `@pre` and `@post` after `@warning` are the additional precondition and postcondition for the robustness.

External and *internal contracts* can be handled with the help of behavioral subtyping. The relationships below should be respected for the consistency of the *external* and *internal contracts* as mentioned in the section 4.2. Namely, the *internal contracts* that represent all behaviors of a function correspond to `supertype` and the *external contracts* that are specialized for behaviors of its intrinsic functionality correspond to `subtype`.

$$
\begin{aligned}
&Pre_{external} \Rightarrow Pre_{internal}\\
&Post_{external} \Leftarrow Post_{internal}
\end{aligned}
$$

The contracts in Case 3 of Fig.9 are translated as

$$
\begin{aligned}
Pre_{external} &= (p \neq NULL)\\
Post_{external} &= (0 \leq \_\_return)\\
Pre_{internal} &= ((p \neq NULL) \vee (p = NULL)) = true\\
Post_{internal} &= ((p \neq NULL) \Rightarrow (0 \leq \_\_return)) \wedge\\
&\qquad ((p = NULL) \Rightarrow (\_\_return = -1)).
\end{aligned}
$$

When the implementation of a function is checked, its *internal contract* should be used. Otherwise, part of the defensive programming becomes dead code for the static checkers. In Fig.9, `return -1;` is dead code under the assumption `p != NULL`. Indeed, in our trial using industrial programs, VARVEL missed an error within the defensive programming part, because it could not reach the error under the assumption of a *external* precondition specified by the programmer.

When the usage of a function by other functions is checked, the choice of contract to be used is up to the users of the static checkers. This means the static checkers should provide an option to check the program against the *external* or *internal contracts*.

6 Handling Re-Entrancy Problem

The use of function-pointers sometimes causes re-entrancy, which, in turn, can manifest new problems related to the following points.

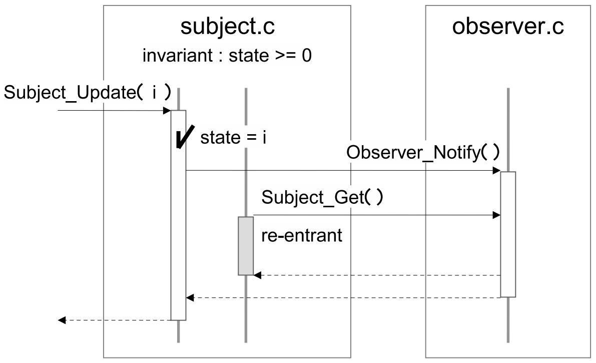
- Lack of global information
- Scope of invariant

Modular checking with the notion of DbC inherently checks one function at a time. It lacks global information along a sequence of function calls, and this can lead to an incorrect result, such as skipping an invariant violation.

In an object-oriented language such as Java, language constructs can be used to define the scope of the invariants. For instance, in JML [14], the scope of the invariants written in a given class is an object of the class, and the invariants are evaluated at the beginning and end of the methods of the object. C is a procedural language and does not have a language construct like Java's class for grouping variables and functions. The scope of the invariants in C is usually global and they must be held at the beginning and end of all the functions.

Figure 10 shows example source codes similar to Subject-Observer pattern [7] to better explain the problem. The function `test` in file `test.c` starts a function call sequence. It first passes the address of function `Subject_Get` in a `subject.c` file to function `Observer_Init` in a `observer.c` file and then calls function `Subject_Update` in `subject.c` with an actual argument `-10`. `Observer_Init` assigns the `Subject_Get` address to a global variable `pGet`, which is a function-pointer. `Subject_Update` assigns its argument `-10` to a global variable `state`, calls function `Observer_Notify` in `observer.c`, and assigns a `0` to a `state` to satisfy the invariant `state >= 0`. Thus, the invariant `state >= 0` is violated during the processing of `Observer_Notify`, which calls back `Subject_Get` through the pointer `pGet`. At the beginning of

`Subject_Get`, the `state` value is `-10` and the invariant `state >= 0` is violated.



subject.c

```
int  state = 0 ;
/**
  @invariant  state >= 0
*/

int  Subject_Get() {
  return  state ;
}

void  Subject_Update( int  i ) {
  state = i ;
  Observer_Notify() ;
  if ( state < 0 )
    state = 0 ;
}
```

observer.c

```
static int  (*pGet)( ) ;
static int  cache = 0 ;

void  Observer_Init( int  (*pFunc)( ) ) {
  pGet = pFunc ;
}

void  Observer_Notify( ) {
  cache = (*pGet)( ) ;
}
```

test.c

```
void  test() {
  Observer_Init( Subject_Get ) ;
  Subject_Update( -10 ) ;
}
```

Fig. 10. Example of Re-Entrancy

To perform modular checking on `subject.c`, a tool checks `Subject_Get` and `Subject_Update` separately and it seems impossible to find the invariant violation previously described, since the context of the sequence from the `test` is not given to it. Instead, the tool warns the invariant violation at the calling point of `Observer_Notify` since it regards the scope of the invariant `state >= 0` as global and evaluates the invariant at the beginning and end of all the functions.

We need the following features to check C programs against invariants in the re-entrancy situation.

- Use of information beyond the module boundary.
- The resolution of an actual function dereferenced by a function-pointer.
- A file-scoped invariant.

Use of Information beyond the Module Boundary SAT-based bounded model checkers including F-Soft perform in-line expansion of all funcsubject.c : Declaration of file-scoped invariant

```
int  state = 0 ;
/**
  @file_invariant  state >= 0
*/
```

subject.c : Realization of *file-scoped invariant* with assumptions, etc.

```
int  Subject_Get() {
  int  r ;
  __assert( state >= 0 ) ;
  __assume( state >= 0 ) ;
  r = state ;
  __assert( state >= 0 ) ;
  __assume( state >= 0 ) ;
  return  r ;
}
```

```
void  Subject_Update( int  i ) {
  __assert( state >= 0 ) ;
  __assume( state >= 0 ) ;
  state = i ;
  Observer_Notify() ;
  if ( state < 0 )
    state = 0 ;
  __assert( state >= 0 ) ;
  __assume( state >= 0 ) ;
}
```

Fig. 11. Example of File-scoped Invariant

tions into one big function. By their nature, such checkers can use source codes information beyond the contract, that is the module boundary. VARVEL can make use of such features provided by F-Soft.

Resolution of Actual Function The re-entrancy in C programs is often caused by a function call through a function-pointer. F-Soft performs a points-to analysis to determine the set of function addresses that are potentially assignable to a function-pointer and constructs a proper CFG in consideration of the function-pointers. VARVEL can make use of such features provided by F-Soft.

File-scoped Invariant Checking the invariants at the beginning and end of all the functions accessible from a static checker cannot detect the invariant violation at the location of re-entrance, because the checker detects the invariant violation at other locations before the re-entrance occurs and hides the invariant violation at re-entrance. Restricting the scope of the invariants to a certain module is necessary. For the C language, one way to represent a module is with a file. To detect the invariant violation at re-entrance, in Fig.10, the scope of the invariant `state >= 0` defined in file `subject.c` should be restricted to `subject.c` itself. We call the invariant of which scope is a file a file-scoped invariant. Figure 11 shows an example of the notation of *file-scoped invariants* (`@file_invariant state >= 0`) and a realization of *file-scoped invariants* by using primitives (assertions, etc.). The realization in Fig.11 corresponds to Case 3 in Fig.3. The *file-scoped invariants* in Case 1 (*entry functions*) and Case 2 (*library functions*) of Fig.3 can also be achieved as well. We experimented with the example in Fig.11 and found that VARVEL can detect the invariant violation at the re-entrance location, which is the beginning of the function `Subject_Get`.

7 Discussions

As mentioned in sections 4,5, and 6, a SAT-based bounded model checker supporting modular checking can overcome the weakness in modular checking if we adopt the notion of behavioral subtyping [15] and enhance DbC notations. However, it needs to consider other aspects for a practical use of the tool. The explanations are in order.

Initialization for invariants. The invariants are met after the initialization is completed. Object oriented languages like C++ and Java have a constructor, which is a language construct for initialization. C dose not have a constructor and the initializer of the variable declaration in C is not expressive enough to initialize the variables with complex data structures. C programmers usually write their own initialization functions. DbC notations must be enhanced to instruct the static checkers about the initialization function.

Scope of invariants for header file. Section 6 explained the scope of the invariants for implementation files (".c"). In the case of header files (".h"), there are at least two interpretations of the file scope. The first is to check the invariants defined in a header file at the beginning and end of the functions declared in the same header file. The second is to check the invariants defined in a header file at the beginning and end of the functions defined in the implementation files that include the header file. Clarification of the relationship between the logical modules and physical containers (files, directories and run-time modules) might lead us to a better choice.

It is not easy to solve the above issues by considering only the syntax and semantics of C language. We need to more thoroughly understand programming convention currently in practical use; how developers write industrial source codes (defensive programming, etc.) and how they create logical modules using language constructs and physical containers. We plan to establish a convention so that the advanced checking discussed above can also be performed by VARVEL.

8 Conclusion

This paper primarily discussed the handling of function-pointers. Although the importance of properly handling function-pointers is well known for the automatic checking of C programs, there are very few checkers that can actually handle them. VARVEL, a SAT-based bounded model checker supporting DbC [16], could be the tool for handling the function-pointers in the modular checking of C programs. We showed how function-pointers can be handled by taking advantage of the notion of behavioral subtyping [15] in example source codes of industrial C programs to which VARVEL has already been applied.

In the future, we will examine the practical conventions of programming and incorporate what we learn into the enhancements to DbC notations and VARVEL.

References

[1] T. Ball and S.K. Rajamani. *The SLAM Project: Debugging System Software via Static Analysis.* In Proc. POPL 2002, pages 1–3, 2002.

[2] D. Beyer, T.A. Henzinger, R. Jhala, and R. Majumdar. *The Software Model Checker Blast: Applications to Software Engineering.* STTT, Vol.9, No.5-6, pages 505–525, 2007.

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

[4] E. Clarke, D. Kroening, and F. Lerda. *A Tool for Checking ANSI-C Programs.* In Proc. TACASf04, pages 168–176, 2004.

[5] D. Detlefs, G. Nelson and J.B. Saxe. *Simplify: A Theorem Prover for Program Checking.* JACM, Vol.52, No.3, pages 365–473, 2005.

[6] E.W. Dijkstra. *A discipline of programming.* Series in Automatic Computation. Prentice Hall Int., 1976.

[7] M. Fahndrich, D. Garbervetsky, and W. Schulte. *A Re-Entrancy Analysis for Object Orietend Programs.* 9th FTfJP at ECOOP 2007.

[8] A. Fehnker, R. Huuck, P. Jayet, M. Lussenburg, and F. Rauch. *Model Checking Software at Compile Time.* 1st IEEE & IFIP TASE 2007.

[9] J.-C. Filliatre and C. Marche. *Multi-prover Verification of C Programs.* In Proc. ICFEMf04, pages 15–29, 2004.

[10] C. Flanagan, K.R.M. Leino, M. Lillibridge, G. Nelson, J. Saxe, and R. Stata. *Extended Static Checking for Java.* In Proc. PLDIf02, page 234–245, June 2002.

[11] R. Funder and M. Felleisen. *Contracts for Higher-Order Functions.* In Proc. ICFPf02, pages 48–59, 2002.

[12] K. Havelund and T. Pressburger. *Model checking JAVA programs using JAVA pathfinder.* STTT. Vol.2, No.4, pages 366–381, 2000.

[13] F. Ivancic, I. Shlyakhter, A. Gupta, M. Ganai, V. Kahlon, C.Wang, and Z. Yang. *Model Checking C Programs Using F-Soft.* In Proc. ICCDf05, 2005.

[14] G.T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Müller, J. Kiniry, P. Chalin, and D.M. Zimmerman. *JML Reference Manual.* 2008.

[15] B. Liskov and J.Wing. *A Behavioral Notion of Subtyping.* In ACM TOPLAS, 16(6), pages 1811–1841, 1994.

[16] B. Meyer. *Applying "Design by Contract".* IEEE Computer, Vol.25, No.10, pages 40–51, 1992.

[17] M.R. Prasad, A. Biere, and A. Gupta. *A Survey of Recent Advances in SAT-Based Formal Verification.* STTT, Vol.7, No.2, pages 156–173, 2005.

[18] Robby, E. Rodriguez, M. Dwyer, and J. Hatcliff. *Checking Strong Specifications Using An Extensible Software Model Checking Framework.* In Proc. TACASf04, pages 404–420, 2004.