Case Study in Specification-Level Defects Detection

Jipeng Wu

Software Institute
Nanjing University

Casual Presentation, 2014

Outline

CodeSpecificationStatechartsModel CheckingExistence Proof

1. 2 types of defects in codes:
   - specification defects
   - implementation defects
2. find a method to detect specification defects
3. prove their existence in current open source projects

CodeSpecificationStatechartsModel CheckingExistence Proof

1. 2 types of defects in codes:
   - specification defects
   - implementation defects
2. find a method to detect specification defects
3. prove their existence in current open source projects

CodeSpecificationStatechartsModel CheckingExistence Proof

1. 2 types of defects in codes:
   - specification defects
   - implementation defects
2. find a method to detect specification defects
3. prove their existence in current open source projects

CodeSpecificationStatechartsModel CheckingExistence Proof

1. 2 types of defects in codes:
   - specification defects
   - implementation defects
2. find a method to detect specification defects
3. prove their existence in current open source projects

CodeSpecificationStatechartsModel CheckingExistence Proof

1. 2 types of defects in codes:
   - specification defects
   - implementation defects
2. find a method to detect specification defects
3. prove their existence in current open source projects

Outline

std def

1. A transition system is defined as a tuple (Q, I, R).
2. Q is the set of states, usually specified by assignments of values to a set ofvariables V;
3. I is a set of states (expressed as predicates on V) defining the initial set of states;
4. R is the transition relation, usually expressed by predicates containing unprimed and primed variables from V for the pre- and post-state.

std def

1. A transition system is defined as a tuple (Q, I, R).
2. Q is the set of states, usually specified by assignments of values to a set ofvariables V;
3. I is a set of states (expressed as predicates on V) defining the initial set of states;
4. R is the transition relation, usually expressed by predicates containing unprimed and primed variables from V for the pre- and post-state.

std def

1. A transition system is defined as a tuple (Q, I, R).
2. Q is the set of states, usually specified by assignments of values to a set ofvariables V;
3. I is a set of states (expressed as predicates on V) defining the initial set of states;
4. R is the transition relation, usually expressed by predicates containing unprimed and primed variables from V for the pre- and post-state.

std def

1. A transition system is defined as a tuple (Q, I, R).
2. Q is the set of states, usually specified by assignments of values to a set ofvariables V;
3. I is a set of states (expressed as predicates on V) defining the initial set of states;
4. R is the transition relation, usually expressed by predicates containing unprimed and primed variables from V for the pre- and post-state.

state explosion

- A flat representation means state explosion.
- Hierarchy

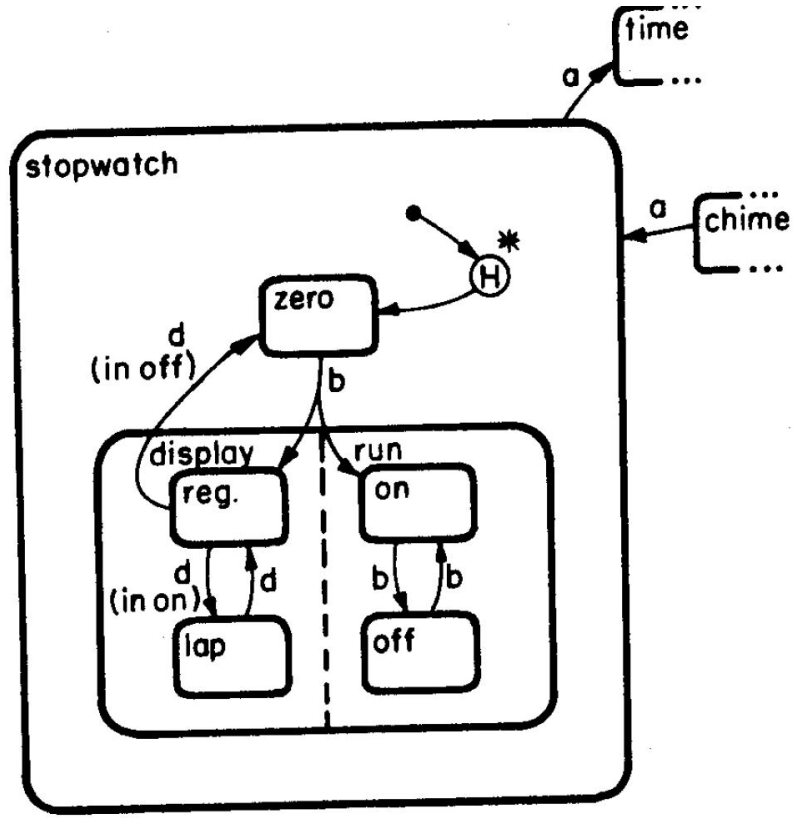


Figure : A sample statecharts with hierarchy

state explosion

- A flat representation means state explosion.
- Hierarchy

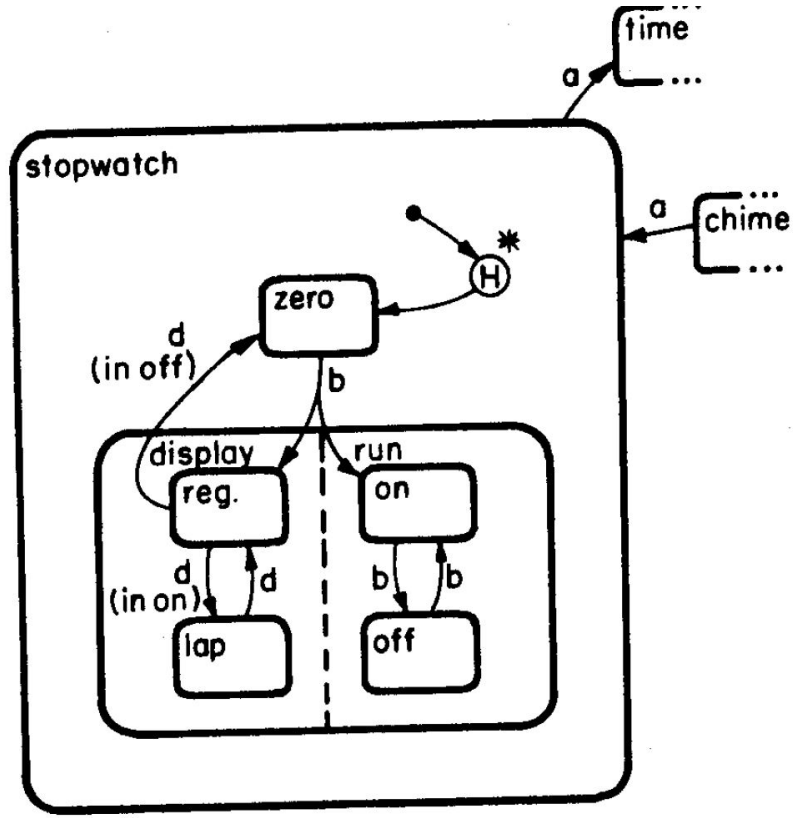


Figure : A sample statecharts with hierarchy

state explosion

- A flat representation means state explosion.
- Hierarchy

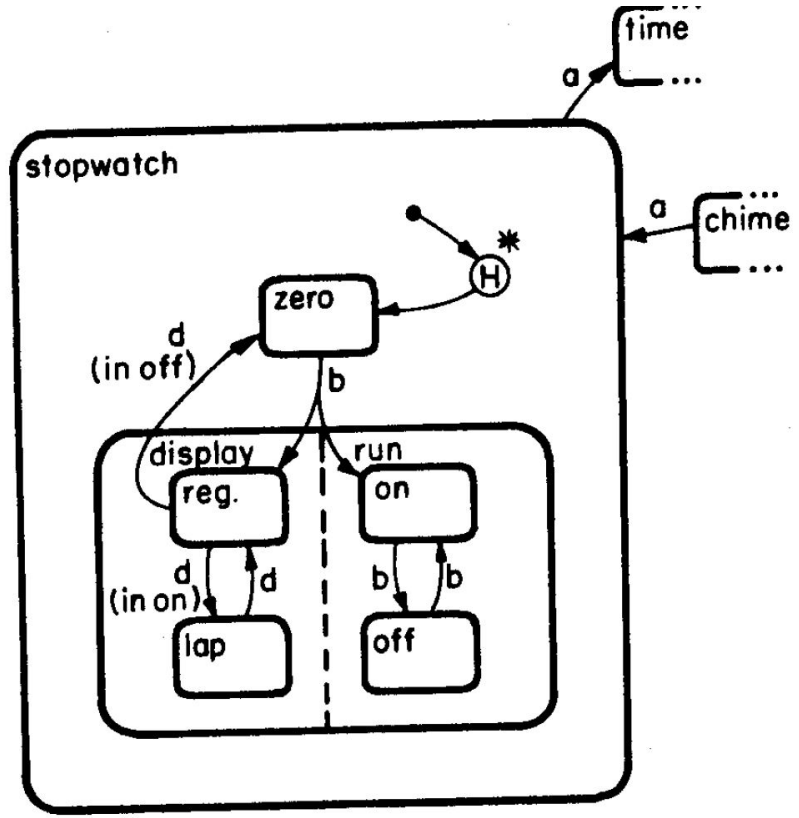


Figure : A sample statecharts with hierarchy

Outline

Difficulty in Reversing Process

1. The hierarchy is invisible for us.
2. We can get a flat representation.

```
class idFile_Memory : public idFile {
    friend class        idFileSystemLocal;

public:
                        idFile_Memory( void );   // file for writing without name
                        idFile_Memory( const char *name ); // file for writing
                        idFile_Memory( const char *name, char *data, int length );  // file for writing
                        idFile_Memory( const char *name, const char *data, int length );   // file for
reading
    virtual                 ~idFile_Memory( void );

    virtual const char * GetName( void ) { return name.c_str(); }
    virtual const char * GetFullPath( void ) { return name.c_str(); }
    virtual int             Read( void *buffer, int len );
    virtual int             Write( const void *buffer, int len );
    virtual int             Length( void );
    virtual ID_TIME_T       Timestamp( void );
    virtual int             Tell( void );
    virtual void        ForceFlush( void );
    virtual void        Flush( void );
    virtual int             Seek( long offset, fsOrigin_t origin );

                        // changes memory file to read only
    virtual void        MakeReadOnly( void );
                        // clear the file
    virtual void        Clear( bool freeMemory = true );
                        // set data for reading
    void                SetData( const char *data, int length );
                        // returns const pointer to the memory buffer
    const char *        GetDataPtr( void ) const { return filePtr; }
                        // set the file granularity
    void                SetGranularity( int g ) { assert( g > 0 ); granularity = g; }

private:
    idStr               name;           // name of the file
    int                 mode;           // open mode
    int                 maxSize;        // maximum size of file
    int                 fileSize;       // size of the file
    int                 allocated;      // allocated size
    int                 granularity;    // file granularity
    char *              filePtr;        // buffer holding the file data
    char *              curPtr;             // current read/write pointer
};
```

Difficulty in Reversing Process

1. The hierarchy is invisible for us.
2. We can get a flat representation.

```
class idFile_Memory : public idFile {
    friend class        idFileSystemLocal;

public:
                        idFile_Memory( void );   // file for writing without name
                        idFile_Memory( const char *name ); // file for writing
                        idFile_Memory( const char *name, char *data, int length );  // file for writing
                        idFile_Memory( const char *name, const char *data, int length );   // file for
reading
    virtual                 ~idFile_Memory( void );

    virtual const char * GetName( void ) { return name.c_str(); }
    virtual const char * GetFullPath( void ) { return name.c_str(); }
    virtual int             Read( void *buffer, int len );
    virtual int             Write( const void *buffer, int len );
    virtual int             Length( void );
    virtual ID_TIME_T       Timestamp( void );
    virtual int             Tell( void );
    virtual void        ForceFlush( void );
    virtual void        Flush( void );
    virtual int             Seek( long offset, fsOrigin_t origin );

                        // changes memory file to read only
    virtual void        MakeReadOnly( void );
                        // clear the file
    virtual void        Clear( bool freeMemory = true );
                        // set data for reading
    void                SetData( const char *data, int length );
                        // returns const pointer to the memory buffer
    const char *        GetDataPtr( void ) const { return filePtr; }
                        // set the file granularity
    void                SetGranularity( int g ) { assert( g > 0 ); granularity = g; }

private:
    idStr               name;           // name of the file
    int                 mode;           // open mode
    int                 maxSize;        // maximum size of file
    int                 fileSize;       // size of the file
    int                 allocated;      // allocated size
    int                 granularity;    // file granularity
    char *              filePtr;        // buffer holding the file data
    char *              curPtr;             // current read/write pointer
};
```

Difficulty in Reversing Process

1. The hierarchy is invisible for us.
2. We can get a flat representation.

```
class idFile_Memory : public idFile {
    friend class        idFileSystemLocal;

public:
                        idFile_Memory( void );   // file for writing without name
                        idFile_Memory( const char *name ); // file for writing
                        idFile_Memory( const char *name, char *data, int length );  // file for writing
                        idFile_Memory( const char *name, const char *data, int length );   // file for
reading
    virtual                 ~idFile_Memory( void );

    virtual const char * GetName( void ) { return name.c_str(); }
    virtual const char * GetFullPath( void ) { return name.c_str(); }
    virtual int             Read( void *buffer, int len );
    virtual int             Write( const void *buffer, int len );
    virtual int             Length( void );
    virtual ID_TIME_T       Timestamp( void );
    virtual int             Tell( void );
    virtual void        ForceFlush( void );
    virtual void        Flush( void );
    virtual int             Seek( long offset, fsOrigin_t origin );

                        // changes memory file to read only
    virtual void        MakeReadOnly( void );
                        // clear the file
    virtual void        Clear( bool freeMemory = true );
                        // set data for reading
    void                SetData( const char *data, int length );
                        // returns const pointer to the memory buffer
    const char *        GetDataPtr( void ) const { return filePtr; }
                        // set the file granularity
    void                SetGranularity( int g ) { assert( g > 0 ); granularity = g; }

private:
    idStr               name;           // name of the file
    int                 mode;           // open mode
    int                 maxSize;        // maximum size of file
    int                 fileSize;       // size of the file
    int                 allocated;      // allocated size
    int                 granularity;    // file granularity
    char *              filePtr;        // buffer holding the file data
    char *              curPtr;             // current read/write pointer
};
```

How to define a hirarchy?

Our Solution is called Specification Defects Detection Using Statecharts.

1. Find out candidate state variables
2. Find out methods with side effects
3. Elicit SDDS Clauses
   1. includes candidate state variables.
   2. The code executed when the clause is satisfied should contain changes to candidate state variables.
4. Convert SDDS Clauses to pre-states

How to define a hirarchy?

Our Solution is called Specification Defects Detection Using Statecharts.

1. Find out candidate state variables
2. Find out methods with side effects
3. Elicit SDDS Clauses
   1. includes candidate state variables.
   2. The code executed when the clause is satisfied should contain changes to candidate state variables.
4. Convert SDDS Clauses to pre-states

How to define a hirarchy?

Our Solution is called Specification Defects Detection Using Statecharts.

1. Find out candidate state variables
2. Find out methods with side effects
3. Elicit SDDS Clauses
   1. includes candidate state variables.
   2. The code executed when the clause is satisfied should contain changes to candidate state variables.
4. Convert SDDS Clauses to pre-states

How to define a hirarchy?

Our Solution is called Specification Defects Detection Using Statecharts.

1. Find out candidate state variables
2. Find out methods with side effects
3. Elicit SDDS Clauses
   1. includes candidate state variables.
   2. The code executed when the clause is satisfied should contain changes to candidate state variables.
4. Convert SDDS Clauses to pre-states

How to define a hirarchy?

Our Solution is called Specification Defects Detection Using Statecharts.

1. Find out candidate state variables
2. Find out methods with side effects
3. Elicit SDDS Clauses
   1. includes candidate state variables.
   2. The code executed when the clause is satisfied should contain changes to candidate state variables.
4. Convert SDDS Clauses to pre-states

How to define a hirarchy?

Our Solution is called Specification Defects Detection Using Statecharts.

1. Find out candidate state variables
2. Find out methods with side effects
3. Elicit SDDS Clauses
   1. includes candidate state variables.
   2. The code executed when the clause is satisfied should contain changes to candidate state variables.
4. Convert SDDS Clauses to pre-states

1. Combine or separate pre-states and rename them
2. Determine initial states
3. Determine the function-level specifications.
4. Generate Statecharts.
5. Statecharts Checking.

1. Combine or separate pre-states and rename them
2. Determine initial states
3. Determine the function-level specifications.
4. Generate Statecharts.
5. Statecharts Checking.

1. Combine or separate pre-states and rename them
2. Determine initial states
3. Determine the function-level specifications.
4. Generate Statecharts.
5. Statecharts Checking.

1. Combine or separate pre-states and rename them
2. Determine initial states
3. Determine the function-level specifications.
4. Generate Statecharts.
5. Statecharts Checking.

1. Combine or separate pre-states and rename them
2. Determine initial states
3. Determine the function-level specifications.
4. Generate Statecharts.
5. Statecharts Checking.

Elicit SDDS Clauses


```
================
idFile_Memory::Read
================
*/
int idFile_Memory::Read( void *buffer, int len ) {

    if ( !( mode & ( 1 << FS_READ ) ) ) {
        common->FatalError( "idFile_Memory::Read: %s not opened in read mode", name.c_str() );
        return 0;
    }

    if ( curPtr + len > filePtr + fileSize ) {
        len = filePtr + fileSize - curPtr;
    }
    memcpy( buffer, curPtr, len );
    curPtr += len;
    return len;
}
```

SDDS Clause of the function *Read*: *!(mode & (1<<FS_READ))*
SDDS Clause of the function *Write*: *!(mode & (1<<FS_WRITE))*，*maxSize!=0*
SDDS Clause of the function *Seek*: *curPtr < filePtr*，*curPtr > filePtr + fileSize*

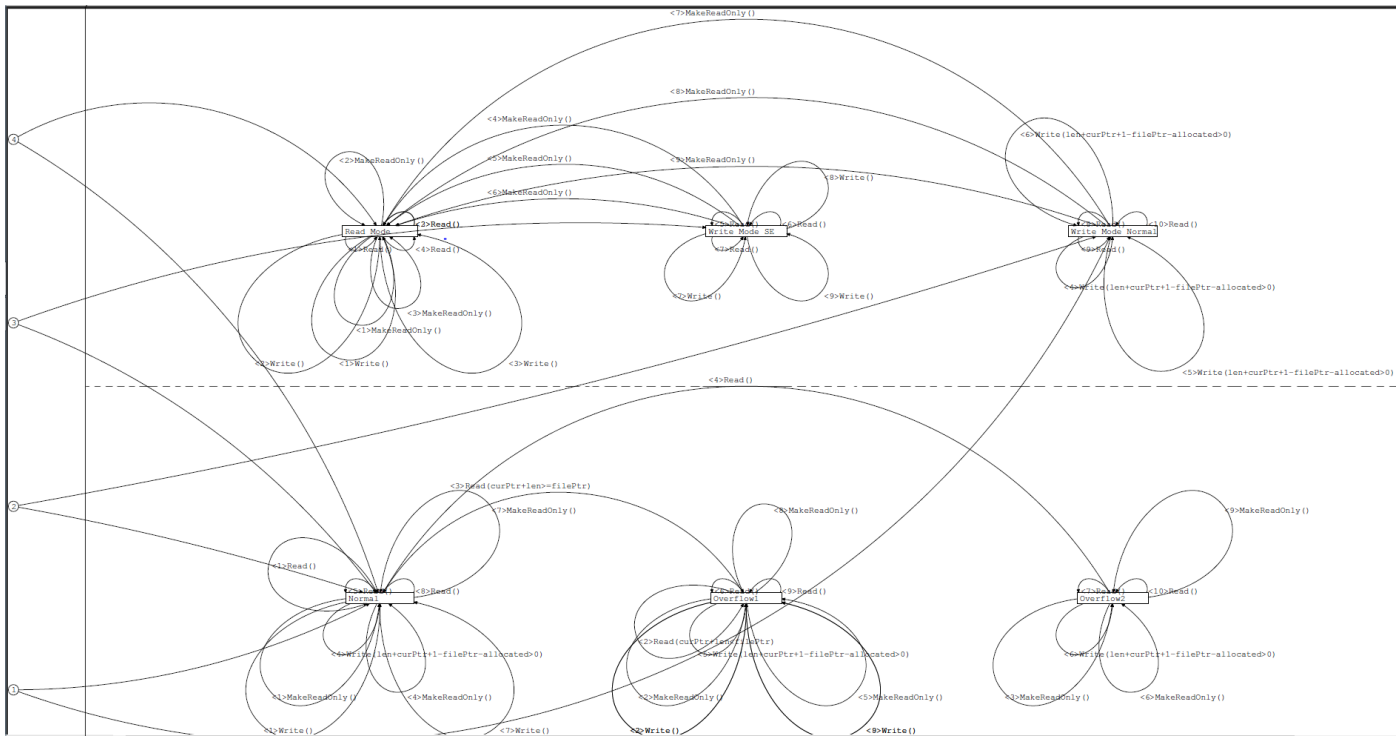
Elicit Pre-states from SDDS Clauses

pre-states of the function *Read*: s_1: mode & (1<<FS_READ); s_2: !(mode & (1<<FS_READ)).
pre-states of the function *Write*: s_3: mode & (1<<FS_WRITE) && maxSize!=0; s_4: mode & (1<<FS_WRITE) && maxSize==0; s_5: !(mode & (1<<FS_WRITE)).
pre-states of the function *Seek*: s_6: curPtr < filePtr; s_7: curPtr > filePtr + fileSize; s_8: curPtr>=filePtr && curPtr<=filePtr+fileSize.

Combine prestates

s_2 includes s_3 and s_4; s_5 includes s_1. Thus we can decompose s_2 to s_3, s_4, s_2-s_3-s_4 and decompose s_5 to s_1, s_5-s_1. s_2-s_3-s_4=s_5-s_1; it is renamed as "Exceptional Mode". s_1 is renamed as "Read Mode". s_3 is renamed as "Write Mode Size Exceeded", s_4 is renamed as "Write Mode Normal"。s_7, s_8, s_9 are parallel to these 3 states; they are respectfully renamed as "Overflow1", "Overflow2", "Normal".

Statecharts



Outline

3 patterns of defects:

1. Pattern 1A state is unreachable. Defects and risks: This state cannot be a prestate of any function, thus there possibly exists defects or correctness-irrelevant redundancy.
2. Pattern 2A common state cannot reach any common states. Defects and risks: This state is actually a exception, which should not be described as a common state in its class specification.
3. Pattern 3There exists null-transitions. Defects and risks: Some conditions may be ignored in the class specification.

3 patterns of defects:

1. Pattern 1A state is unreachable. Defects and risks: This state cannot be a prestate of any function, thus there possibly exists defects or correctness-irrelevant redundancy.
2. Pattern 2A common state cannot reach any common states. Defects and risks: This state is actually a exception, which should not be described as a common state in its class specification.
3. Pattern 3There exists null-transitions. Defects and risks: Some conditions may be ignored in the class specification.

3 patterns of defects:

1. Pattern 1A state is unreachable. Defects and risks: This state cannot be a prestate of any function, thus there possibly exists defects or correctness-irrelevant redundancy.
2. Pattern 2A common state cannot reach any common states. Defects and risks: This state is actually a exception, which should not be described as a common state in its class specification.
3. Pattern 3There exists null-transitions. Defects and risks: Some conditions may be ignored in the class specification.

Q & A