CONCURRENT ERROR DETECTION OF PROGRAM EXECUTION BASED ON STATECHART SPECIFICATION

I. Majzik, J. Jávorszky, A. Pataricza and E. Selényi[1]

In common procedural and object-oriented programming languages the control flow of a program is a hierarchical structure. Concurrent error detectors proposed in previous works were able to check the sequence of statements in procedures and the procedure return addresses, but the problem of checking the allowed sequence of procedure calls and inter-process synchronization remained unsolved. In this paper we propose an approach of checking the higher-level program control flow using reference information based on UML statechart specification.

1. Introduction

Fault tolerance is a key design factor in computer systems providing critical services. In the majority of applications such redundancy techniques are required, that are general purpose, fit to common, commercial system components and development processes and at the same time improve the availability of the system significantly. In such systems, *watchdog processors* (WP, [1]) are often used for concurrent detection of control flow errors. A WP compares the run-time and the reference control flow represented in a compact form by *signatures*, symbolic labels of the program state. In common procedural and object-oriented programming languages the control flow of a program is a hierarchical structure. The sequence of processes, the sequence of procedures in processes and finally the sequence of statements in procedures can be distinguished. Watchdog processor methods proposed in the literature and elaborated in previous works were able to check the sequence of statements and the return addresses of procedure calls. The problems of checking the allowed sequence of procedure calls and the inter-process synchronization remained unsolved.

[1] Dept. of Measurement and Information Systems, Technical University of Budapest,
Műegyetem rkp. 9., H-1521 Budapest, Hungary, E-mail: [majzik, javor, pataric, selenyi]@mit.bme.hu

The widespread use of specification languages describing concurrent, multi-process systems enable the checking of the previously uncovered higher levels of the control flow. UML, the Unified Modeling Language is the most recent standard for specification, visualization and documentation of (object-oriented) systems [5]. Its behavioral diagrams like sequence, collaboration and statechart diagrams describe exactly the allowed sequence of procedure calls and the inter-process communication and synchronization. Since these diagrams form the basis of the automatic code generation, they can also be used as the reference for checking the correct control flow.

In this paper we propose a method for checking the sequence of procedure calls. It allows the integration of previous methods of statement level checking with the higher level checks. The reference information used for checking the sequence of procedure calls is derived automatically, using the UML description of the system behavior. It has the advantages that the same description is used for checking as used for the (automatic) code generation, this way the checks are performed against the *specification*, and errors in code generation/modification not reflected in the specification can also be highlighted.

2. Concept of the procedure level checking

In our approach, we utilize the modular structure of the WP proposed in [2] and integrate the checks at different levels as individual modules. The statement level checking is performed by a hardware WP using assigned run-time signatures and embedded reference ones as described by the SEIS (Signature Encoded Instruction Stream) scheme in [2]. On the procedure level, the checking is based on assigned run-time signatures (called here *procedure identifiers*) and a stored reference database. It can be pointed out that the time overhead of the checking on the procedure level is not critical since the relative frequency of checking is low (once for each procedure call). Accordingly, the procedure level module can be implemented in software. The following preprocessing steps are required for each process to support this software based self-checking:

1. The statechart specification of the process is analyzed and the reference table of the procedure call sequence is derived (see Section 3). It is stored in a textual/tabular form.

2. The *statement level preprocessor* analyzes the source code of the process (or the active object), identifies the statements and assigns signatures to them according to the SEIS scheme.

3. The *procedure level preprocessor* identifies the procedures, assigns unique identifiers to them and modifies the program source as follows. First, it inserts the source of the checker function that includes the static reference table of the procedure calls (derived in Step 1). Moreover, it inserts a call of the checker function as the first statement of each procedure.

The preprocessed (modified) source code is compiled and linked using the standard compiler of the language. The statement sequence and the return addresses of procedures are checked in run time by the hardware WP. The sequence of procedures is checked in software. When a procedure is started, it calls the checker function parameterized with its own identifier. The checker function contains the reference table as a static array (inspired by the approach of [4]) and looks up whether the actual procedure identifier is a valid successor of the previous one. If not, then an error is detected. . The first procedure after the start of the program is accepted automatically.

3. Automatic derivation of the reference information

The reference information is derived automatically using the statechart specification of the process (active object). The allowed sequences of procedures are given by the possible sequences of transitions in the statechart. To derive the reference table of the procedure calls, the statechart has to be processed, including (i) the resolution of the state hierarchy and (ii) the resolution of concurrent procedures. As an illustration of the approach, we present an example by analyzing the statechart of Figure 1.

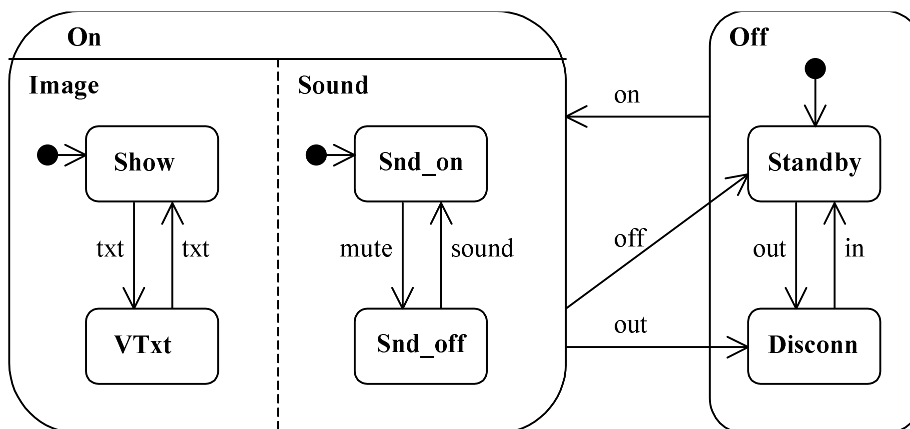

Figure 1: Statechart of a TV controller

- Resolution of the state hierarchy. Transitions between basic states (being at the lowest level of the state hierarchy) are kept unaltered, while transitions emanating from or pointing to a composite state are replaced by transitions connecting the embedded *basic* states only. I.e. a transition pointing to a composite state is replaced by a transition which points to the basic initial state(s) of the embedded diagram(s). Similarly, if a transition emanates from a composite state then it is replaced by a set of transitions emanating from the basic states of the (concurrent) composite state. E.g., the transition *on* pointing to state *On* in Figure 1 is replaced by a set of transitions emanating from *Standby* and *Disconn* and reaching both *Show* and *Snd_on*.

- Resolution of concurrent transitions. In the case of transitions in concurrent regions of a composite state, all interleaving sequences of transitions are allowed. E.g. transition *txt* of region *Image* is independent from transitions *mute* and *sound* of region *Sound*. Accordingly, the allowed predecessor-successor pairs of these transitions are the original (*txt, txt*), (*mute, sound*), (*sound, mute*) and the interleaving ones as (*txt, mute*), (*txt, sound*), (*mute, txt*) and (*sound, txt*).

The result of these steps is the adjacency matrix of state transitions, i.e. procedures (Table 1).

	txt	**mute**	**sound**	**out**	**in**	**on**	**off**
txt	1	1	1	1			1
mute	1		1	1			1
sound	1	1		1			1
out					1	1	
in				1		1	
on	1	1		1			1
off				1		1	

Table 1: Reference table of the TV controller

According to the reference matrix and assuming the encoding of the procedures in the order of their appearance in the header of Table 1, the implementation of the checker function and the first row of procedure *mute* (inserted by the preprocessor) are as follows:

```
unsigned char last=UNKNOWN;
int procedure_checker(unsigned char actual) {
    static unsigned char reference_table[7][7]={
        1, 1, 1, 1, 0, 0, 1,
        1, 0, 1, 1, 0, 0, 1,
        1, 1, 0, 1, 0, 0, 1,
        0, 0, 0, 0, 1, 1, 0,
        0, 0, 0, 1, 0, 1, 0,
        1, 1, 0, 1, 0, 0, 1,
```

```
        0, 0, 0, 1, 0, 1, 0
    }
    if ((last!=UNKNOWN) && (!reference_table[last][actual]) {
        error();
    }
    last=actual;
}

int mute() {
    procedure_checker(1);
    /* Procedure code comes here */
}
```

The function `error()` may invoke the termination of the erroneous process and/or invoking error recovery functions. The first procedure after the start of the program is accepted automatically, as the reference value (the identifier of the previous procedure) is set to `UNKNOWN`.

4. Conclusion

In our opinion, our approach of checking the higher-level program control flow is a proper combination of low-cost, high-speed hardware based checking (using embedded reference information at the statement level) and software based self-checking (using reference tables based on UML statechart specification at the procedure level). In our experimental implementation, the statecharts of processes/objects are extracted from the design database of the UML CASE tool Innovator [3] and a PL/SQL program derives the reference matrix of the allowed procedure calls. The checker functions and the necessary calls are inserted into the C program source by the modified version of the SEIS WP preprocessor. Since the UML specification of the application can describe also the interaction (synchronization and communication) of processes, the reference information for a process level checker module can be extracted in a similar way. The implementation of this checker is a task of our future research.

References

1. MAHMOOD, A.; McCLUSKEY, E. J. Concurrent error detection using watchdog processors - A survey. IEEE Trans. on Comp., 37(2):160-174, 1988.
2. MAJZIK, I.; HOHL, W.; PATARICZA, A.; SIEH, V. Multiprocessor checking using watchdog processors. International Journal of Computer Systems - Science & Engineering, 11(5):125-132, September 1996.
3. MID. Innovator v6.1, http://www.mid.de/e/innovato/index.htm, 1998.
4. SIEH, V.; HOENIG, J. Software-based concurrent control flow checking. Internal report 10/95, FAU Erlangen-Nuremberg, IMMDIII, December 1995.
5. Rational Software et al. UML Summary, version 1.1, http://www.rational.com, 1997.