Incorporating Resource Safety Verification to Executable Model-based Development for Embedded Systems

Jianliang Yi, Honguk Woo, James C. Browne, Aloysius K. Mok
Dept. of Computer Sciences
The University of Texas at Austin
{jlyi, honguk, browne, mok}@cs.utexas.edu

Fei Xie
Dept. of Computer Science
Portland State University
xie@cs.pdx.edu

Ella Atkins
Dept. of Aerospace Engineering
University of Michigan
ematkins@umich.edu

Chan-Gun Lee
Dept. of Computer Engineering
Chung-Ang University
cglee@cau.ac.kr

Abstract

This paper formulates and illustrates the integration of resource safety verification into a design methodology for development of verified and robust real-time embedded systems. Resource-related concerns are not closely linked with current xUML model-based software development although they are critical for embedded systems. We describe how to integrate resource analysis techniques into the early phase of an xUML-based development cycle. Our hybrid framework for resource safety verification combines static resource analysis and runtime monitoring. A case study based on an embedded controller for satellite simulation, TableSat, illustrates the benefits obtained by incorporating resource verification into design and combining static analysis and runtime monitoring.

1. Introduction

Model-based development has been recognized as a practical method for efficiently developing correct and robust control-oriented real-time embedded systems. Generally model-based development has focused on verification and testing for functional or timeliness aspects. However, embedded software also involves para-functional resource-related aspects, termed *resource (bound) properties* in this paper, such as enforcing resource limits on CPU time, memory, battery power, network bandwidth, etc. Yet verification and testing for such resource properties has been rarely addressed in executable model-based development. Resource-related language constructs are not incorporated in the action semantics of executable models since early design is intended to be platform independent. Accordingly resource properties have not been linked with functional verification from the beginning of the development cycle. This limitation often renders the process of **resource safety verification** (the verification of resource bound properties) *non-systemic* or *ad hoc* at best leading to excessive cost for monitoring and analysis. Furthermore, resource safety verification is usually deferred to testing during/after the implementation phase. Resource safety violations detected during implementation testing commonly require redesign and reimplementation of the system.

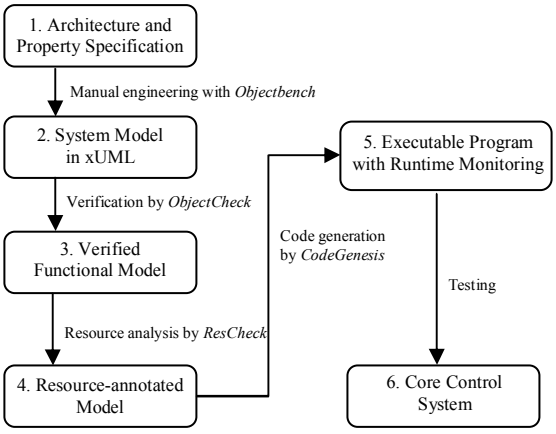

Figure 1. Development Cycle

In this paper, we propose a software engineering discipline which incorporates resource safety verification into a design and development methodology for embedded systems (Figure 1). Specifically we show how to incorporate the verification of para-functional resource properties into the software development cycle of executable model-based approaches. The methodology integrates:
(a) xUML (eXecutable Unified Modeling Language [1]) modeling and simulation-based model testing processes supported by the commercial software modeling and simulation environment *Objectbench* [2]
(b) Automatic code generation from xUML models by *CodeGenesis* code generator [4]
(c) Formal functional verification by model checking for xUML models [3]

1080-1812/08 $25.00 © 2008 IEEE
DOI 10.1109/RTAS.2008.28

(d) Resource bound checking based on static analysis and dynamic monitoring [7, 8]

ObjectCheck [3] is used to validate the xUML model with respect to selected functional properties while the resource verifier for embedded systems, named *ResCheck*, deals with resource properties to provide the comprehensive autonomous support for resource safety verification. The development cycle of Figure 1 includes the following steps:

- **Architecture and property specification.** The system is mapped to an architecture where the software is partitioned into core functionalities and none-core functionalities. The goal is to design a core segment of software which is sufficiently compact so as to be amenable to formal analysis both by functional verifiers and runtime monitoring methods. Our model-based approach concentrates on the development of the core segment.
- **xUML model.** The executable model in xUML for the core segment is manually developed by using *Objectbench*. *Objectbench* is a software development tool that supports a UML-like OOA method [1] and generates executable and compilable specifications for analysis models. The model is executed and tested under the control of the discrete event simulator in *Objectbench*.
- **Verified functional model.** The model is formally verified with respect to the selected functional properties by the xUML model checking tool *ObjectCheck*. The xUML specification of the model is automatically translated to the S/R model specification to be verified by COSPAN model checker [5]. The details of xUML model checking can be found in [3].
- **Resource-annotated model.** The model is further specified by *manually* adding resource-related specification into the functional specifications. At present our approach requires the developers to write the partial implementation code of resource-related operations and insert it to the executable state actions of the state model in xUML. Language constructs to provide the appropriate abstraction of resource specification in xUML models are currently in progress. The model is then analyzed by *ResCheck*. As a result, violations of resource properties can be detected at design time and the model revised accordingly. Runtime monitoring code for resource safety verification is *automatically* inserted in the action specifications in the model for those resource properties that cannot be statically verified. By using a low overhead runtime monitoring scheme, *ResCheck* supports efficient dynamic evaluation of resource consumption upper bounds. The detailed procedure for static analysis and runtime monitoring in *ResCheck* is explained in Section 3 and 4.
- **Executable program.** The model is automatically translated into executable program code via *CodeGenesis* code generator [4] for a particular target platform. Based on the resource-annotated model specification derived from the static resource analysis in the previous step, the generated program contains the minimum necessary monitoring code. Then the program is run and tested on the target platform. Runtime violations of resource properties can be detected and reported. Resource safety violation rates above critical thresholds may require redesign.
- **Core control system.** The model satisfying all the functional and resource properties is ready to be extended with the further design and implementation for non-core functionalities. It is important to note that the development cycle in Figure 1 can be iterated until the software implementation for the core segment is verified by model checking (the task spanning box 2 to 3 in Figure 1) and testing with runtime monitoring (the task spanning from the box 4, 5 to 6).

In the development methodology above, verification for functional or resource properties is done by combining model checking, resource analysis and runtime monitoring. As mentioned, using xUML model checking for functional properties has been well studied in the previous work [3]. Therefore we address a complementary problem, *efficient runtime monitoring*, considering resource properties as our primary concern for verification. Note that due to their inherently dynamic nature, resource properties are not usually addressed in the static-verification-only context.

Our work primarily aims at incorporating the verification process for resource-related properties into the early development cycle, thereby lowering development cost and enhancing the quality of resource critical embedded software. Resource safety verification requires additional resource-related specification in the model as input to further analysis at the design phase. It is worthwhile to note, however, that in our approach, the developer's labor can be reduced by exploiting the executable semantics of xUML and the autonomous tool support for resource analysis and monitoring code insertion. The executable semantics of xUML enables the model specification to be tested in the simulation environment and to be automatically translated into an executable program [1, 3], and furthermore it allows resource-related operations to be specified as part of state actions in the state model.

Our approach for resource safety verification employs a *hybrid* framework where static analysis and monitoring techniques work cooperatively. The runtime monitoring in *ResCheck* makes explicit use of static analysis results to cope with the possible performance overhead of traditional runtime monitoring mechanism. The static resource analysis in *ResCheck* translates an executable

model containing resource-related code into a tree-based resource evaluation structure. Note that the resource evaluation structure may involve statically-unbounded variables e.g., loop bounds that can only be dynamically determined. This often renders the verification inherently incomplete at analysis time and necessitates runtime monitoring support. For monitoring efficiency, the static resource analysis simplifies runtime operations by having in-lined monitoring code that collects only selected high-level runtime information (for updating the statically-unbounded variables) and conducts simple arithmetic calculation in the tree-based resource evaluation structure with the collected information (for updating the resource consumption upper bounds). Our runtime monitoring relies on static analysis and thus monitors discrete updates of a small set of specific variables in the program execution, instead of directly tracking and managing dynamic resource usage information. Since in practice testing alone cannot completely guarantee system correctness, it is natural that a product system employs runtime monitoring support in the execution environment as part of exception handling. This would in turn incur inordinate performance overhead to the execution environment unless the monitoring algorithm is carefully designed. The hybrid framework — specifically, lightweight runtime monitoring based on static analysis results — addresses this problem of runtime overhead. Moreover, it can cover a wide variety of property types in software safety requirements.

The critical functionalities identified in a system design can be verified by model checking and/or completely tested in the development cycle above, and then the corresponding software component can be treated as a core segment. Our approach has been successfully tested for modeling and developing the architecture of a practical embedded system, an embedded controller for satellite simulation (in Section 4). The hybrid, lightweight monitoring technique is also naturally consistent with desirable extension to the implementation and integration of non-core segment where static verification may be inherently limited. The monitoring technique for providing the safe interplay between core and non-core segments of a control-oriented embedded system is one of our future research directions.

The rest of this paper is organized as follows: Section 2 reviews some previous work including xUML model checking and resource bound verification. Section 3 describes our approach for incorporating the resource safety verification to xUML model-based embedded system development. Then Section 4 provides the case study with detail examples and evaluations, and Section 5 concludes and suggests future research.

2. Related Work

2.1. Model Checking for xUML

xUML is a commercially supported object-oriented modeling language which can specify action semantics additionally. Xie et al. [3] present a tool *ObjectCheck*, enabling a model checker to verify a software system modeled by xUML. They find that extant model checkers cannot directly support object-oriented modeling languages due to syntactic and semantic differences. They propose a solution for this by providing an automatic translation technique, which is briefly introduced following. In addition, they address large state space issues by presenting a space reduction algorithm. In *ObjectCheck*, designers of the system use the Property Specification Interface and xUML Visual Modeler to specify the properties of the system and xUML model. An xUML-to-S/R translator converts them to S/R query and S/R model respectively. The COSPAN model checker [5] accepts these inputs and checks whether the query is valid in the model. In case that the verification fails, COSPAN model checker generates an error track and then the Error Report Generator produces an error report in xUML from the error track. To help the debugging process, Error Visualizer creates a test case from the error report and reproduces the error by running the xUML model with the test case.

2.2. Resource Bound Verification

There have been many efforts toward providing the capability of resource bound checking for program codes. Ajay et al. [7] classifies the extant work into static, dynamic, and hybrid approaches. The dynamic approaches have an advantage in that they can be implemented easily; however, they introduce extra run-time overhead for monitoring. On the other hand, static approaches do not impose the run-time overhead. But they depend on program analysis requiring complex implementations and sometimes they cannot be done completely [7]. Furthermore if resource bounds depend on dynamic data which are known only during runtime, then pure static analysis may fail to check the resource bounds. Recently Ajay et al. [7] and Mok et al. [8] proposed novel techniques combining the static and dynamic approaches. They are referred to as hybrid approaches. In the following, we introduce major research results for bound checking and highlight the unique features of *ResCheck*. Crary and Weirich's work [6] is a purely static approach. It uses virtual clocks to augment a program and certifies that the program does not exceed its resource boundary by showing that the virtual clocks cannot expire. Dynamic approaches include [9, 10, 11, 12]. Most of the research taking the dynamic approach supports Java and converts the byte code of the source program to a functionally equivalent code appended with run-time monitoring for resources. Since most languages lack fine-grained resource control mechanism, many dynamic approaches provide resource management interface so that users can

design various policies. The work in [8] focuses on statically certifying the resource usage. Run-time monitoring is only used to guarantee the validity of user-provided information. Once the information is determined to be invalid, the execution is simply terminated. *ResCheck* uses static analysis to establish a resource usage evaluation model, which is used to estimate the resource consumption at runtime when more accurate information is dynamically obtained.

One of the critical resource properties for embedded system is timing constraints. A variety of WCET computation strategies e.g., path-based [13], tree-based [14], and IPET-based [15, 16] have been proposed. In traditional WCET analysis [17, 18], the following restrictions are made: (1) no recursion, (2) no function pointers, (3) the upper bound of each loop has to be known. Generally loop bounds can be treated as undetermined factors in dynamic environments, and resource consumption bounds can be updated at runtime as more accurate information about such undetermined factors is collected. Two-step WCET analysis for reusable and portable code is proposed in [19]; the first step computes the abstract WCET information and the second step uses the abstract information to compute concrete WCET bounds in a specific context. This approach is referred to as distributed WCET computation [20]. The tree-based, retargetable approach in [14] separates WCET analysis and WCET evaluation, which is in a similar direction to our design for resource evaluation. However, this previous work neither addresses the statically-unknown bound problem nor considers dynamic evaluation mechanism.

3. Overview of Resource Safety Verification

In a critical embedded system, resource properties are often considered as important as functional properties for system correctness. Upon the emergence of platform-independent languages that enable a single system to be deployed on several different platforms, explicit *separation* between resource analysis and resource evaluation has accordingly been investigated [14, 19, 20]. This separation has been usually intended to address the platform-independent design and implementation in that resource analysis creates a parameterized resource consumption skeleton for which the instantiation and evaluation occurs at deployment time by binding the parameters to specific platform-dependent configuration values. This separation-based approach can be extended to the temporal dimension for coping with environmental dynamic nature. During the execution of an embedded system, the environment settings might continuously change and the system might need to be reconfigured accordingly. In general, capturing all the situations in a dynamic environment by a single static model would lead to insufficient accuracy. Of course, "pure" runtime monitoring (that monitors executions without any help from static analysis) can be used in this case to enforce resource properties, but normally at the price of significant overhead. Therefore we formulate the resource evaluation/verification for an embedded system into the runtime monitoring problem, and exploits static resource analysis techniques (on the xUML model) to minimize runtime monitoring overhead.

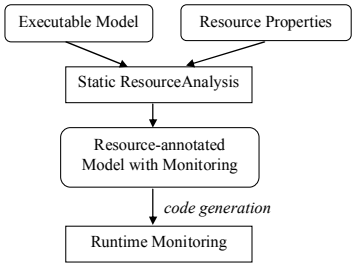

Figure 2. *ResCheck* Architecture Overview

The architecture of *ResCheck* is shown in Figure 2. *ResCheck* supports (1) static resource analysis for an executable model in xUML and (2) runtime monitoring for a program generated from the model. Note that code generation is done by using the generic architecture template of *CodeGenesis* [4]. The primary goal of *ResCheck* is (1) to detect possible resource property violation early in the system design and development phases, thus allowing rapid prototyping of the core software functionalities, and (2) to provide efficient runtime monitoring support for the cases where the safety decision on the system model cannot be completely made by static analysis.

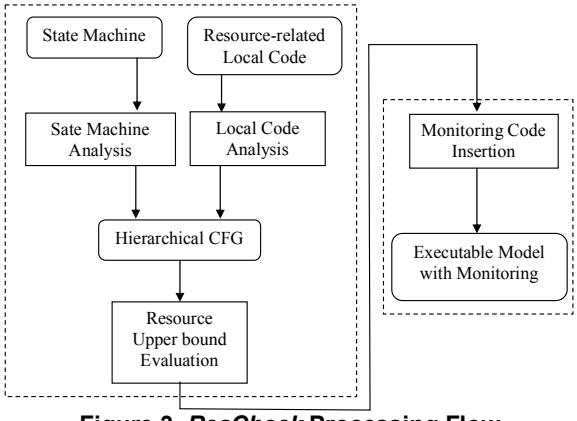

Figure 3. *ResCheck* Processing Flow

Figure 3 illustrates the processing flow of *ResCheck*, the detail of which is described in the following subsections.

3.1. Static Resource Analysis

Given an executable model with resource-related code, *ResCheck* establishes an evaluation mechanism for resource consumption at analysis time, by executing three

tasks: state machine analysis, local code analysis, and resource (consumption) upper bound evaluation.

State machine analysis. This analysis is applied on state machines in a xUML state model [1]. Each state machine is treated as a Control Flow Graph (CFG) and transformed into a hierarchical CFG where loop and condition structures are abstracted out. Starting from a given CFG, if a loop structure is found in the CFG, it is replaced by a loop node. Then a sub-CFG is created and associated with the loop node to represent the states in the loop structure. This abstraction is repeated recursively until a hierarchical CFG where each sub-CFG contains no loop structure is constructed. Without loop structure, each sub-CFG can be expanded to a tree. Calculating resource consumption in a tree is very efficient. Conditional structures can be processed in a similar way. If a condition structure is found, it is replaced by a conditional node. Each branch is represented by a sub-CFG and is associated with the conditional node. This abstraction for conditional structures can reduce the exponential growth of the number of paths that must be evaluated, thus reducing the resource consumption evaluation cost.

Local code analysis. Executable actions associated with each state in hierarchical CFGs must be analyzed. In doing so, we first construct hierarchical CFGs for the actions. We use a term *local code* to denote analysis-required actions in a state because we limit our consideration to specific languages, e.g., C or Java, for action specification in xUML. Since local code maintains directly usable hierarchical structure information, when parsing the local code, a simple translation is needed for hierarchical CFG construction. By combining these local hierarchical CFGs to the corresponding state machine hierarchical CFG, finally we can construct a hierarchical CFG capturing both state transitions and state actions.

Resource upper bound evaluation. Resource consumption evaluation is based on a hierarchical CFG. Here we consider an accumulative resource type as an example and notate its upper bound estimation by `RES`. Assume that the `RES` of basic blocks (i.e., the terminal nodes of a hierarchical CFG) is calculated and known. Since there is no loop in any sub-CFG, the `RES` of the sub-CFG is equivalent to that of the corresponding expanded tree. Given multiple paths in the tree, the maximum value is given as the `RES` of the sub-CFG. For a loop node with loop bound `lp` and loop body `cfg_lb`, the `RES` of the loop node is given by `lp * RES(cfg_lb)`. For a conditional node with branch condition `C` and branches cfg_br_i, the `RES` is given by $RES(C) + \max\{RES(cfg\_br_i)\}$. The evaluation is hierarchical and tree-based, thus efficient.

Traditional static resource analysis is performed on the code after the final system is developed. In contrast, we apply static analysis one step earlier, on the xUML model with partial implementation of related local code. If any resource property violation is detected at this early stage, the model can be accordingly modified. Then the new model goes through the static analysis step again, until no definite violation is found by the resource upper bound evaluation.

3.2. Runtime Monitoring

The resource upper bound evaluation at analysis time does not necessarily complete resource safety verification since the statically-analyzed model can have resource-related statically-unbounded variables. We term such variables that require monitoring during the program execution, *dynamic bounds*. Often a loop bound cannot be precisely determined at analysis time but can be determined only during the runtime execution; and the loop body might contain heavily resource-related operations, e.g., a real-time messaging software may have different resource restrictions for encoding/decoding messages depending on the actual runtime environment, so the maximum possible number of messages per connection or the maximum possible length of a single message cannot be precisely known *a priori*, and then must be treated as dynamic bounds. In general, developer-provided estimates for such dynamic bounds would be used to accomplish the initial evaluation. But this easily could be too pessimistic or optimistic, leading to inaccurate evaluation and verification.

Monitoring code insertion. In *ResCheck*, runtime checking for dynamic bounds and resource evaluation calls are automatically inserted into the local code in state actions. For example, for a dynamic loop bound, the loop counter is inserted into the loop to check whether or not the current loop bound is valid. If the actual loop exceeds the bound, the bound is expanded and the resource upper bound evaluation function will be called by this bound update. Resource properties are verified by this dynamic resource evaluation mechanism, and if violations occur, the violations can be detected by the new evaluation.

Comparing with pure runtime monitoring mechanism, the runtime monitoring in *ResCheck* has several advantages from automatically generated monitoring code. First the number of monitoring operations can be safely reduced by exploiting static analysis results. Moreover, the cost of a single monitoring operation is even minimized since the monitoring code simplifies runtime resource monitoring to comparison checking for dynamic bounds.

Consider the example program in Figure 4(a), and assume there is no resource overrun during the execution. Here suppose the function `AllocMem(n)` to perform `n` bytes memory allocation. For a pure monitoring system in figure(b), `CheckMem(n)` needs to be called to check if the allocation of `n` bytes is safe at each memory allocation by `AllocMem(n)`; so $200N$ times of `CheckMem` operations can be required. Assume that the outmost loop bound N is unknown at static analysis time, and so it is a dynamic

bound here. Static analysis can find that the inner loop consumes (100+200)*100 bytes. Then the outmost loop needs to be monitored to control the total memory allocation (in figure(c)), and so the total number of checks significantly reduces to *N*. Furthermore, while for a pure monitoring system, each runtime monitoring operation involves an execution of the `CheckMem` function (that often relies on underlying system support), in *ResCheck*, it corresponds to the simple arithmetic operation, that is, increasing the loop count and comparing the count with the current bound. Therefore we can obtain lower overhead for runtime monitoring by *ResCheck*.

```
While (ReadData() != NULL) {
    for (int j = 0; j < 100; j ++) {
        AllocMem (100);
        ...
        AllocMem (200);
    }
}
```
(a)

```
While (ReadData() != NULL) {
    for (int j = 0; j < 100; j ++) {
        CheckMem (100);
        AllocMem (100);
        ...
        CheckMem (200);
        AllocMem (200);
    }
}
```
(b)

```
int loopCount = 0;
While (ReadData() != NULL) {
    if (++ loopCount > BOUND)
        evaluation();
    for (int j = 0; j < 100; j ++) {
        AllocMem (100);
        ...
        AllocMem (200);
    }
}
```
(c)

Figure 4. Example Programs

Since it is hard to recover the damage when resource usage overrun has occurred and resources have been consumed, last-minute violation detection by a pure monitoring system is not attractive. *ResCheck*, in contrast, can detect possible violations before they happen and therefore leave margin for recovery and error processing. Our runtime evaluation scheme is thus flexible in providing fault tolerance against resource violations. Consider the example in Figure 5 to illustrate this benefit. First the program reads in some configuration value to `conf`. Then the program enters a loop to input, process and store data. Suppose here `AllocMem(conf * conf)` asks for more memory than the limit and violates some resource property. In a pure monitoring system, the violation can be caught right before the allocation, after inputting data. In this case, since the data has already been consumed yet no output is generated, the appropriate recovery step from such inconsistent state should be taken. In *ResCheck*, the violation detection can happen as early as right after getting the configuration value.

```
int conf = ReadConf();
/* Monitor the value of conf */
while () {
       ReadData();
       AllocMem(conf * conf);
       ProcessData();
       StoreData();
}
```

Figure 5. Example Program

4. Case Study

In this section, a real-world application, the simulator for control-oriented embedded satellite systems provided by the Space System Laboratory in University of Michigan, is used to illustrate our approach.

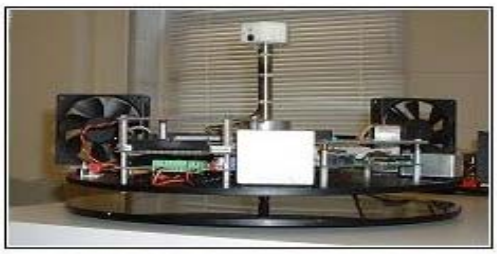

Figure 6. TableSat platform

4.1. TableSat: Control Simulator

TableSat [21] is a one degree-of-freedom "tabletop" satellite that emulates to the extent possible the dynamics, sensing, and actuation capabilities of a spacecraft. Consistent with conventional control systems, TableSat (as illustrated in Figure 6) is made up of the following hardware components: sensors (4 sun sensors, a gyroscope, and a magnetometer), processor (a Diamond systems Prometheus PC/104 board with QNX operating system), network (802.11b wireless network interface), and thrusters (two computer fans providing clockwise and counterclockwise torques). TablesSat onboard software is composed of the following 4 threads, each of which executes as a real-time task running periodically at constant frequency:

- *State Estimator* thread that performs sensor readings and estimates the current state of TableSat
- *Controller* thread that applies control laws to calculate the input voltages of the two computer fans
- *Actuator* thread that sets the input voltages of the fans
- *Communication* thread that supports data and command transmission from/to an external client program

4.2. Modeling in xUML

Currently we are focused on modeling the structure of onboard TableSat software threads and the concurrency mechanism for accessing global data, with the intention of building an extensible framework for various experiment scenarios. The TableSat xUML model developed in *Objectbench* specifically includes:

- The class model that depicts the definitions and relationships of threads and global data
- The state model that depicts the behavior of the threads and the concurrency mechanism

The class model contains:

- 4 classes for TableSat threads: `TS_ESTIMATOR, TS_CONTROLLER, TS_ACTUATOR, TS_COMMUNICATION`
- 7 classes for global data being shared by the above threads: `GD_ESTIMATOR, GD_STATEESTIMATOR,`

GD_CONTROLLER, GD_ACTUATOR, GD_FANDATA, GD_SENSORREADINGS, GD_STATUS

- A scheduler class with periodic time intervals for TableSat threads: SCHEDULER
- A lock interface class to the global data being shared by the threads: GD_LOCK_INTERFACE

The behavior of a class over time is formalized in a state machine where a transition between states is triggered by an event. An event can be generated within either an inter- or intra-class relation. For example, the state model for TS_ESTIMATOR (*State Estimator* thread) class looks like the simplified one in Figure 7 where we describe the partial actions only in Ready and ReadSensors states.

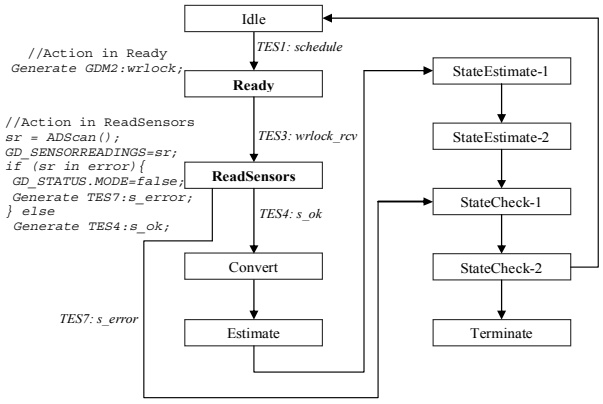

Figure 7. State Model of TS_ESTIMATOR

For example, the actions of the ReadSensors, Convert, Estimate, and StateEstimate1-2 states in the figure correspond to the respective primary functions of *State Estimator* thread: (1) performing A/D scan of sensors, (2) converting sensor readings into engineering units, (3) filtering sensor data according to the specified estimation strategy, (4) loading the filtered data to the global data structure for the Controller thread. Each action accesses different global data and so needs to communicate with different lock interface instances.

The xUML model execution proceeds as follows: SCHEDULER class advances the current time and periodically generates schedule events to TableSat threads according to the specified time intervals for cyclic thread executions. For example, when a schedule event is received by an instance of the TS_ESTIMATOR class, the instance performs the associated action in the current Ready state, that is, generating wrlock event to GD_LOCK_INTERFACE to update GD_SENSORREADINGS class value after conducting A/D sensor scan. Note that in the xUML model execution, only a single action for a given state machine can be in execution at any time during model execution. A set of simultaneously enabled actions in different state machines are executed by the simulator at the same simulation time in a random order.

4.3. Examples by *ResCheck*

ResCheck is responsible for resource property verification on a state model. If any possible resource property violation is detected by static analysis, the state model needs to be revised and the resource property is re-examined, until no resource property violation can be detected by static analysis.

We use the TS_ESTIMATOR class as an example in this section. One resource property to be enforced on the TS_ESTIMATOR state model is that the total amount of memory consumption by the *State Estimator* thread is less than the given MEM_MAX bytes.

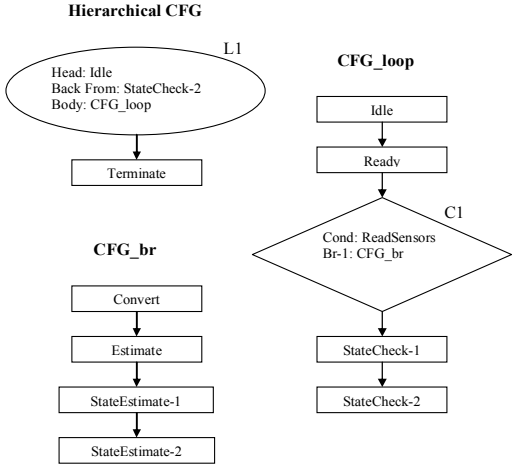

Figure 8. Hierarchical CFG of the State Model

The state machine for the *State Estimator* thread is in Figure 7. As described in Section 3, the state machine is treated as a CFG, and the corresponding hierarchical CFG, which is shown in Figure 8, is constructed during static analysis. A loop structure *L1* is detected at the top level, within which a conditional structure *C1* is found. Each rectangular node in Figure 8 is a state node, which can be associated with local code.

The TS_ESTIMATOR class needs to construct a sparse matrix and input some sensor data in the ReadSensors state. The corresponding functions are named MakeSparse and ReadData. These two functions are the main source of memory consumption. In order for accurate memory consumption analysis and monitoring, the detailed code must be available. After analyzing MakeSparse and ReadData functions as local code, which will be demonstrated below, their hierarchical CFGs are associated with the corresponding state node in Figure 8. This step constructs a final hierarchical CFG which captures both state transition and relevant local operations.

By evaluating the final hierarchical CFG, the memory consumption in local code is first evaluated, and is combined at the state machine level. The evaluation

captures the memory consumption behavior of the whole state machine, including related local operations.

```
void MakeSparse(int * data) {
    int i, j, pos = 0;
    CV * ptr, * tmp;

    for (i = 0; i < ROW; i ++) {                    /* L1 */
        sm.nvals[i] = data[pos ++];
        if (sm.nvals[i] == 0) sm.colVals[i] = NULL;  /* C1 */
        else {
            ptr = NULL;
            for (j = 0; j < sm.nvals[i]; j ++) {     /* L2 */
                tmp = (CV *)malloc(sizeof(CV));      /* B7 */
                tmp -> col = data[pos ++];
                tmp -> val = data[pos ++];
                tmp -> next = NULL;
                if (ptr == NULL) sm.colVals[i] = tmp;/* C2 */
                else ptr -> next = tmp;
                ptr = tmp;
            }
        }
    }
}

void ReadData()
{
    char * ptr, * tmp;
    int i, flag = 1, counter = 0;

    while (flag) {
        ptr = ReadBuf();
        if (* ptr != 0) {
            ptr ++;
            for (i = 0; i < FIELD_NO; i ++) {
                tmp = malloc(FIELD_SIZE);
                bcopy(ptr, tmp, FIELD_SIZE);
                sd[counter].field[i] = (double *)tmp;
            }
            counter ++;
        } else
            flag = 0;
    }
}
```

Figure 9. Program *MakeSparse* and *ReadData*

To illustrate the local code analysis, we use the `MakeSparse` function as an example. The `MakeSparse` function constructs a sparse matrix from given data. The code is shown in Figure 9. The number of rows in the matrix is fixed (defined as `ROW` in the example). Each row is stored as a linked list. Each node of the linked list represents a non-zero element in that row and contains the column number and the value. The code of `ReadData` is also shown in Figure 9.

We use memory consumption for the `MakeSparse` function to illustrate the processing procedure of *ResCheck*. The hierarchical CFG of the `MakeSparse` function is constructed during static analysis, as shown in Figure 10. CFG3, CFG5 and CFG6 are not included in the figure because they are very simple one-node CFGs. The memory consumption occurs in the basic block B7, which contains the `malloc` function call. Every time B7 is executed, a fixed number of bytes (12 bytes to be specific) are allocated to store one element in the sparse matrix.

The loop bound of the outside loop *L1*, which is equal to the number of rows in the sparse matrix, can be determined at analysis time since this number is fixed. But the loop bound of the inside loop *L2* cannot be determined at analysis time. It depends on the sensor data and system configuration at runtime, and can change widely in extreme cases. A static bound that covers all circumstances could be very loose. The strategy of *ResCheck* is to start with a relatively tight bound, which is provided by the developer, for normal situations, and to rely on runtime monitoring to evaluate the resource consumption upper bound in the extreme cases, to guarantee total resource consumption stays within the limit.

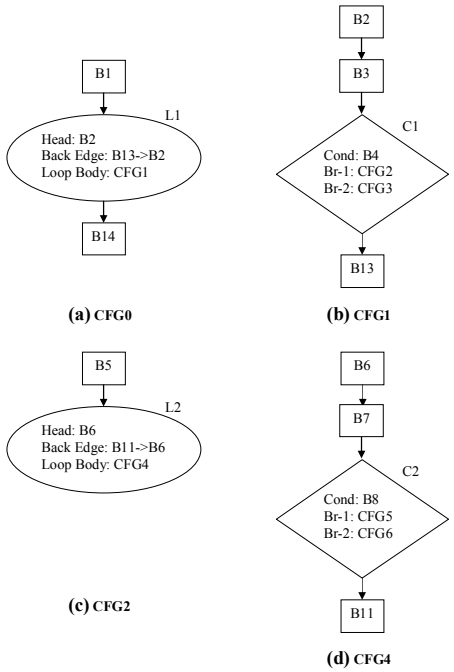

Figure 10. Hierarchical CFG of *MakeSparse*

During the initial evaluation of memory consumption on `MakeSparse`, the hierarchical CFG in Figure 10 can be further reduced to the hierarchical CFG in Figure 11, based on which the runtime evaluation is performed.

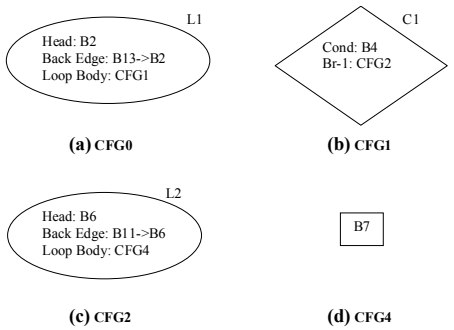

Figure 11. Reduced Hierarchical CFG

The runtime monitoring code is inserted into the original code, as shown in Figure 12 (surrounded by rectangular boxes). The variable `_lb_L10_` stores the current estimated bound of the inner loop, and is

initialized to the developer-provided loop bound (denoted as `INIT_VAL`). The variable `_lc_L10_` is the loop counter. Once `_lc_L10_` exceeds `_lb_L10_`, the validity of the previous resource consumption evaluation no longer holds. The loop bound needs to be expanded and a new evaluation is performed at runtime. If the new evaluation suggests possible memory consumption overrun, the program will be terminated. Otherwise, the program is still safe to continue execution.

```
void MakeSparse(int * data)
{
    int i, j, pos = 0;
    CV * ptr, * tmp;

    for (i = 0; i < ROW; i ++) {
        sm.nvals[i] = data[pos ++];
        if (sm.nvals[i] == 0) sm.colVals[i] = NULL;
        else {
            ptr = NULL;
            Static int _lb_L10_ = INIT_VAL;  // Loop Bound Initialization
            int _lc_L10_ = 0;      // Loop Counter Initialization
            for (j = 0; j < sm.nvals[i]; j ++) {
                if (++ _lc_L10_ > _lb_L10_) { // Counter Update & Check
                    _lb_L10_ += _lb_L10_;     // Bound Expansion
                    // Runtime Evaluation
                    EvaluateBound("H-MakeSparse", "L10", _lb_L10_);
                }
                tmp = (CV *)malloc(sizeof(CV));
                tmp -> col = data[pos ++];
                tmp -> val = data[pos ++];
                tmp -> next = NULL;
                if (ptr == NULL) sm.colVals[i] = tmp;
                else ptr -> next = tmp;
                ptr = tmp;
            }
        }
    }
}
```

Figure 12. *MakeSparse* with Runtime Monitoring

Now consider the performance of *ResCheck* monitoring. In the following we use the `MakeSparse` function and `ReadData` function as the examples and consider the memory usage as the monitored property. With pure monitoring mechanism, in order to enforce the limit of memory usage, a check function is necessary before each `malloc` function call to check if the following `malloc` function call will push the total amount of consumed memory over the limit. This is the main source of overhead in pure monitoring. In *ResCheck*, the runtime monitoring mechanism is inserted into the original code, as shown in Figure 12. The main overhead is introduced at runtime from two aspects: loop bound checking and resource consumption evaluation.

The experiment results for performance overhead are in Table 1 and Table 2. In the tables, the columns #Chk, #Eva, and ExTim denote the number of runtime checks, the number of resource consumption evaluations and the overall execution time respectively. For the `MakeSparse` function, the numbers of runtime checks are the same for pure monitoring and *ResCheck* monitoring. The overhead of the version with pure monitoring is more than 4%, while that with monitoring and evaluation in *ResCheck* is less than 2%. In this example, *ResCheck* reduces the monitoring cost by simplifying resource checking into loop bound checking.

Table 1. Performance of *MakeSparse*
(Number of executions: 10^4)

	#Chk(10^4)	#Eva	ExTim (s)	Overhead
Original Code	0	0	1.063	---
Pure Monitoring	1000	0	1.108	4.23%
ResCheck	1000	4	1.084	1.98%

In our current design, the bound is expanded in the way that a certain percent of margin is left, e.g., exponentially. This strategy can minimize the number of runtime resource consumption evaluation as shown in the above table. For the `MakeSparse` example, the execution time per evaluation is $5.958 * 10^{-6}$ s. Furthermore the evaluation overhead is amortized to many execution times over time.

For the `ReadData` function, the inner loop bound is fixed. So only the outside loop needs to be monitored in *ResCheck*. This reduces the number of runtime checks per function execution to 16, comparing with 75 in pure runtime monitoring. This further helps to reduce the monitoring overhead in *ResCheck*. In our experiment, the overhead of *ResCheck* is only 0.51%, while the overhead of pure monitoring is more than 3%.

Table 2. Performance of *ReadData*
(Number of executions: 10^5)

	#Chk(10^4)	#Eva	ExTim (s)	Overhead
Original Code	0	0	1.564	---
Pure Monitoring	75	0	1.617	3.39%
ResCheck	16	3	1.572	0.51%

5. Conclusion

This paper presents and illustrates an xUML model-based embedded software development method which integrates a runtime verification scheme for resource safety verification into a complete development methodology. We use static resource analysis on executable state models in xUML containing resource-related code to enable systematic construction of low overhead runtime monitoring. Our method is conceptually consistent with today's hybrid verifiers that combine static analysis and formal verification for functional properties and runtime monitoring for other non-verifiable properties. Our study shows the advantages of such a hybrid verifier at the design (xUML) that explicitly uses resource analysis techniques in the context of implementing and testing a resource critical embedded system. The current implementation has achieved a high degree of automation but not complete automation. Work is ongoing to fully complete the automation of the translation from the xUML specification to conform to the input specification format of *ResCheck*.

There are several topics we have identified as future work. It is desirable to adapt runtime verification in

core/non-core embedded system architecture [22]. The present method focuses on the safe implementation of core components by iteratively using model checking, static analysis, and runtime monitoring within a combined toolset. Incorporating non-core components in this method may require *black-box monitoring* where in- and out-stream of relatively unreliable components can be input to runtime verification only in a limited way. We are also interested in developing an *adaptive* policy for controlling valid parametric resource bounds during runtime evaluation of resource consumption. It would be possible to formulate the problem of developing such a runtime adaptive policy into that of dynamic bound adjustment where multiple uncertain variables are involved in some inequality expression. The update patterns of the bounds and the pre-calculable influence weight for each variable over overall resource consumption can be used for providing the flexibility between violation detection timeliness and false alarm rates. Finally, we will extend the methodology to span multiple implementations of core components to enable continued safe operation with degraded resource configurations.

6. Acknowledgement

This research was supported in part by the National Science Foundation under Grant Number 0613665 "Collaborative Research: SoD-TEAM: A Feedback-Based Architecture for Highly Reliable Embedded Software", and by the grant CR070019 from Seoul R&BD Program. This research was also supported in part by grants of computer software from QNX Software Systems and HyPerformix, Inc.

7. References

[1] S. J. Mellor and M. J. Balcer, *Executable UML: A Foundation for Model-Driven Architecture,* Addison-Wesley, New York, 2002.

[2] SES: *Objectbench User Reference Manual*, SES, 1996.

[3] F. Xie, V. Levin and J. C. Browne, "ObjectCheck: A Model Checking Tool for Executable Object-Oriented Software System Designs", *Proc. of FASE*, 2002.

[4] SES: *Code Genesis Manual*, SES, 1996.

[5] R. H. Hardin, Z. Har'El and R. P. Kurshan, "COSPAN", *Proc. of Computer Aided Verification*, 1996.

[6] K. Crary and S. Weirich, "Resource Bound Certification", *Proc. of Symposium on Principles of Programming Languages*, 2000.

[7] C. Ajay, E. David and I. Nayeem, "Enforcing Resource Bounds via Static Verification of Dynamic Checks", *ACM Transactions on Programming Languages and Systems (TOPLAS)*, volume 29 , issue 5, 2007.

[8] A. K. Mok and W. Yu, "TINMAN: A Resource Bound Security Checking System for Mobile Code", *Proc. of the European Symposium on Research in Computer Security (ESORICS)*, 2002.

[9] L. Gong, G. Ellison and M. Dageforde, *Inside Java 2 Platform Security 2nd ed: Architecture, API Design, and Implementation*, Addison-Wesley, 1999.

[10] M. Kim, S. Kannan, I. Lee and O. Sokolsky, "Java-MaC: a Run-Time Assurance Tool for Java Programs", *Electronic Notes in Theoretical Computer Science 55***,** 2001.

[11] W. Binder and J. Hulaas, "A Portable CPU-Management Framework for Java", *IEEE Internet Computing*, 8(5):74-83, 2004.

[12] G. Czajkowski and T. Eicken, "JRes: A Resource Accounting Interface for Java", *Proc. of OOPSLA*, 1998.

[13] F. Stappert and P. Altenbernd, "Complete Worst-Case Execution Time Analysis of Straight-line Hard Real-Time Programs", *Journal of Systems Architecture*, 46(4):339-355, 2000.

[14] A. Colin and I. Puaut, "A Modular and Retargetable Framework for Tree-Based WCET Analysis", *Proc. of ECRTS*, 2001.

[15] P. Puschner and A. Schedl, "Computing Maximum Task Execution Times – a Graph-Based Approach", *Journal of Real-Time Systems*, 13(1):67-91, 1997.

[16] Y. Li and S. Malik, "Performance analysis of embedded software using implicit path enumeration", *Proc. of the ACM SIGPLAN Workshop on Languages, Compilers, & Tools for Real-Time Systems*, 1995.

[17] P. Puschner and C. Koza, "Calculating the maximum execution time of real-time programs", *Real-Time System*, 1(2):159-176, 1989.

[18] M. Schoeberl and R. Pedersen, "WCET Analysis for a Java Processor", *Proc. of JTRES*, 2006.

[19] P. Puschner and G. Bernat, "WCET Analysis of Reusable Portable Code", *Proc. of ECRTS*, 2001.

[20] N. Aissa, C. Rippert and G. Grimaud, "Distributing the WCET computation for embedded operating systems", In *Proc. of RTSS, Work in Progress Session*, 2004.

[21] M. F. Vess, "System Modeling and Controller Design for a Single Degree of Freedom Spacecraft Simulator", *Master Thesis*, University of Maryland, 2005.

[22] S. Kowshik, G. Rosu, and L. Sha, "Static Analysis to Enforce Safe Value Flow in Embedded Control Systems", *Proc. of DSN*, 2006.