Using formal methods for the development of safe application-specific RTOS for automotive systems

Kabland Toussaint Gautier Tigori, Jean-Luc Béchennec, Sébastien Faucou, Olivier Roux

Using formal methods for the development of safe application-specific RTOS for automotive systems.

Kabland Toussaint Gautier Tigori [1], Jean-Luc Béchennec[2], Sébastien Faucou[3], Olivier Henri Roux[1]
LUNAM Université. CNRS[2], École Centrale de Nantes[1], Université de Nantes[3]
IRCCyN UMR 6597 (Institut de Recherche en Communications et Cybernétique de Nantes),
Nantes, FRANCE.

***Abstract*—This paper focuses on the development of system software for resource constrained embedded systems such as automotive systems. In these systems, the real-time operating system (RTOS) needs to be highly configurable and tailorable according to the application requirements, the dependability constraints, and the hardware constraints. In this paper, we propose a safe solution to this challenge. We describe a method to generate application-specific RTOS based on formal models. This method allows to verify the correctness of the generated RTOS and to guarantee that it does not contain dead code.**

I. Introduction

A. Context

During the past 15 years, the increase of software based functions provided in vehicles has provoked a paradigm shift, from the federated architecture (one function per Electronic Control Unit (ECU)) to the integrated architecture (several functions per ECU based on more powerful microcontrollers). This evolution has been formalized with the release of AUTOSAR (*AUTomotive Open System Architecture* [2]). At the same time, ensuring the functional safety of these systems has been recognised as a major concern, so the ISO 26262 standard has been established.

AUTOSAR defines a layered architecture, where each layer is populated with components. These components are very configurable so as to be usable with a broad variety of requirements. If we focus on functional safety, one component is of primary importance: the real-time operating system (RTOS), because it is responsible for managing the execution of the other components on the shared hardware platform.

Different challenges arise when implementing an AUTOSAR OS [2] compliant component: 1) it must be configurable to meet a broad variety of application requirements; 2) its design must provide evidences of its safety, as these evidences are required to write the safety arguments of the systems that use it; 3) being used in an embedded context, the component must be tailorable to meet the resource constraints usually imposed by ECU hardware platforms, such as limited memory footprint and CPU overhead.

To tackle all these challenges, implementations of AUTOSAR OS components are usually obtained from a software product line. Unfortunately, these software product lines do not allow to easily address the safety requirements. For instance, most of them use the C preprocessor to perform at least a part of the configuration. In this context, it becomes difficult to apply static analysis tools to verify the correctness of the configured component.

In this paper, we propose an alternative approach, based on formal methods. The software product line that we describe allows to automatically generate the configured component from formal models, and at the same time to verify its correctness against a set of safety properties. It also provides a guarantee that the configured component contains no dead code.

B. Related Works

Several studies have been dedicated to the specialization of operating systems according to application-specific requirements.

In [9], aspect oriented programming is used for the development of a configurable operating system. This approach consists in a precise encapsulation of each part of the operating system in an aspect. More precisely, each aspect implements a part of the operating system that provides exactly one feature. The overall operating system is obtained by weaving the aspects corresponding to the required features. In this work, no solution is provided to verify the correctness of the generated component.

In [5], a configuration technique based on a software-component library called *DREAMS* is presented. DREAMS offers a set of components that can be customized at source code level and configured according to the application requirements with a configuration tool called *TEReCS*. The system to be generated is specified by two graphs: a *Resource graph* which represents the hardware components with their available connections in the system; and a *Process communication graph* which describes the communication behavior of the application (the communication between each process in the application). In addition to this, a *Universal Resource Service graph* describes the library of all available components. Then, these three graphs are composed to form a *Resource Service Graph* which describes all the required services and resources of the system. Eventually, the configured component is generated from this graph.

In [7], a technique of customization for operating system services by combining two approaches is presented. The first approach consists in using an object-oriented operating system architecture that relies on composition to facilitate code reuse and configuration [8]. The second approach proceeds by putting a minimal set of trusted functions in the kernel protection domain and all remaining operating system functionalities in the user-level domain [1]. The final kernel is minimal but for this approach, the application developer must be an operating

system expert since he should be able to manipulate low-level operating system mechanisms.

C. Contributions and outline

In this paper, we present a new approach based on formal methods for the automatic generation of safe application-specific RTOS. From an existing RTOS, a formal model is built. This model implements all the features and observable behaviours of the RTOS. A model of the application layer is built and combined with the RTOS model in order to produce a model of the overall software system. A reachability analysis is performed on this model in order to detect all infeasible paths that correspond to the part of the operating system code that is never executed (also called dead code). These paths are removed in order to obtain a model that supports only the features used by the application. Our approach being based on formal methods, it is possible to verify the correctness of the generated component with static analysis techniques. In our approach, we propose to use a model-checker to verify the model against a set of logic formulas that capture the expected behaviour of the RTOS. Lastly, the entire source code of the application-specific RTOS is generated from the model.

The paper is organized as follows: section II presents the problem of a configuration process based on the C preprocessor; section III describes a model-based approach for application specific configuration of RTOS; section IV presents the model that have been built from the source code of Trampoline RTOS [3], an AUTOSAR and OSEK compliant operating system; section V explains how verification is performed; the final section concludes the paper.

II. Problem analysis

In automotive embedded systems, the RTOS are designed to be highly configurable for the reasons exposed in the previous section. In current practices, it is common to make an extensive use of the C preprocessor to achieve this goal.

A. Why configuration based on the C preprocessor does not scale?

Consider the excerpt of a function of the Trampoline RTOS shown in figure 1. Within this function, `#if` statements are used to select or dismiss pieces of code at compile time when some condition is true. For instance, all the code presented in figure 1 will be compiled in the RTOS object code if `TASK_COUNT > 0` is true. This kind of implementation seems unsafe for several reasons.

First, the preprocessor does not enforce strict syntaxic and semantic rules, so it is quite easy to make a mistake in the expression of the condition. Second, the mix of preprocessor statements and C code decrease the readability of the code and thus the efficiency of code reviews. Quoting [9] "*this approach does not scale -it quickly leads to* #ifdef hell *and a bad separation of concerns*". This is also a problem for static analysis tools: state-of-the-art tools such as Frama-C [6] need to preprocess the files before to analyze them and thus are not able to detect mistyped expressions in the preprocessor code.

The consequences of a mistyped preprocessor statement can range from the insertion of dead code in the RTOS object code, to erroneous algorithms. Erroneous algorithms might be detected by subsequent verification and validation steps. However, only static analysis techniques can provide strong evidences of correctness. Dead code is much more difficult to detect and should not be considered as inoffensive. It impacts the memory footprint of the RTOS, and its runtime performances as well (by impacting the i-cache and the pipeline states). Moreover, it could become a serious problem from a security point-of-view.

Going back to our example, one can notice that even if the preprocessor directives are correct, this implementation does not tailor perfectly the RTOS. In a static system, it is possible to know at compile-time if the result of the test `(result = (tpl_status)E_OK_AND_SCHEDULE)` is always false. In this case, the inner branch is never taken: it is dead code. If this situation occurs for a lot of functions, the amount of dead code can become important. Unfortunately, detecting such a problem requires to perform a semantic interpretation of the system, which is out of reach of the preprocessor.

```
FUNC(StatusType, OS_CODE) tpl_activate_task_service(
  CONST(tpl_task_id, AUTOMATIC) task_id)
{
    ...
#if TASK_COUNT > 0
  IF_NO_EXTENDED_ERROR(result)
    result = tpl_activate_task(task_id);
    if (result == (tpl_status)E_OK_AND_SCHEDULE)
    {
      tpl_schedule_from_running();
# if WITH_SYSTEM_CALL == NO
      if (tpl_kern.need_switch != NO_NEED_SWITCH)
      {
        tpl_switch_context(
          &(tpl_kern.s_old->context),
          &(tpl_kern.s_running->context)
        );
      }
# endif
    }
  IF_NO_EXTENDED_ERROR_END()
#endif
    ...
}
```

Fig. 1. An excerpt of tpl_activate_task_service function

B. Why model-based approach is promising?

As an alternative to preprocessor-based tools, we propose to use model-based methods. In these methods, the configuration is performed on a model, which is then used to generate the code of the configured component. We advocate that these methods are promising, for several reasons.

First, when the model is executable, it provides the possibility to simulate the system.

Second, in the case of a static system and if the model is designed with care, static analysis can be used to detect the branches that are never used. Then, it is straightforward to prune these branches from the model. The resulting model can be used to generate a source code where all instructions are useful. The corresponding object code is smaller and exhibits better performances.

Third, static analysis techniques can also be used to verify the correctness of the pruned model. This verification should

ensure that no expected behaviours of the RTOS have been broken during the configuration process, thus increasing the confidence in the correctness of the generated component.

III. Our approach

Our approach is illustrated by figure 2.

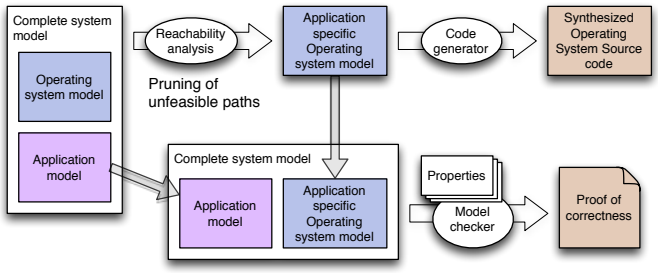

Fig. 2. Application-specific RTOS generation and verification process.

Modelling: In a first step, a model of the system must be established. It is obtained by combining a detailed model of the RTOS and an abstract model of the application requirements. To write these models, we use networks of finite automata extended with variables. The variables are bounded integers, so as to ensure that the models have a finite state space. The model of the RTOS is written once and re-used for every configuration. The variables and the imperative code blocks of the RTOS are an integrated part of this model. More informations on this model are given in section IV. Conversely, the model of the application requirements is defined for each configuration. In this model, for each tasks of the application, an extended finite automaton is provided. This automaton describe all the possible sequences of calls to RTOS API functions performed by the tasks. The model of the system is obtained by synchronizing the automata of the model of the RTOS and the automata of the model of the application requirements.

Reachability analysis: In a second step, a set of reachability analysis are performed in order to detect unreachable states in the RTOS model. When a state is unreachable, its incoming and outgoing instructions can be removed. Once all these states and transitions have been removed, the resulting model contains only the parts of the RTOS that are really used by the application. Removed parts can be as small as a branch in a function and as big as a service, *ie.* a set of functions with their associated data structures. The last work consists in extracting the configured RTOS model from the model of the system.

This whole step is carried by our tool written in Python, that uses the UPPAAL model checker `VerifyTA` (despite the fact that for the moment, our models are untimed). The pruned model computed by the tool can now be used as an input to the remaining steps: verification and code generation. Code generation is briefly described in the next paragraph, whereas verification is described with more details in section V.

Code generation: As explained in the next section, the model of the RTOS is a *code level* model: transitions are associated with sequential code blocks, nodes are associated with branching. Given that some branches have been pruned, some sequential code blocks can be merged. Then, the code generation is straightforward. Our code generator reads the label of the transitions in the model and outputs the corresponding C code in the relevant files.

IV. RTOS Model

Trampoline is an open-source AUTOSAR OS compliant RTOS developed by the *Real-Time Systems* group at IRCCyN in Nantes, France. Initially developed for an academic use, it has also been transferred to industry and used in production.

Following AUTOSAR OS standard, Trampoline is a static RTOS (all objects are created at compile time) dedicated to control-command systems. It is mainly written in C and, as illustrated in figure 1, it makes an extensive use of the preprocessor. The current version contains 174 functions for a total of 4530 lines of code.

The model of Trampoline has been built within UPPAAL [4], an integrated environment for the modelling and verification of timed systems. An UPPAAL model is a network of timed automata communicating over synchronous channels and extended with variables and functions. Each function of Trampoline is modeled by an automaton. The discrete structure of this automaton corresponds to the control flow graph of the function. Each transition is attached to a set of instructions corresponding to a basic block of the source code of the function. Thus, all the branches and all C statements of the source code of Trampoline are present in the model. Each location corresponds either to a test or to a function call. Test are translated in guards over the control variables. Function calls are mimicked by a two step mechanism. In a first step, the caller automaton releases the execution of the callee by taking a transition guarded by a synchronization over a channel. When this transition is taken, a shared variable is set to 0. The caller is blocked in the target location of the transition while the shared variable equals to 0. In a second time, after the execution of the function, the callee sets the shared variable to 1 to release the caller. This mechanism allows to obtain a sequential execution conforming to the real execution.

Fig.3 shows the model of `tpl_activate_task_service` function (see figure 1). This function calls others functions such as `tpl_activate_task` that activates a task and inserts it into the list of ready task and `tpl_schedule_from_running` that performs a rescheduling if needed. Double circle represents initial location and each location describes the function execution state. A transition can be labelled by a synchronisation (channel name with "`?`" or "`!`"), a guard (comma separated logical terms) and an assignment (comma separated assignments by "`:=`").

Let us use this example to illustrate the function call mechanism. The call to `tpl_activate_task` is emitted through a synchronization over channel `tpl_activate_task`. Once the call has been emitted, the automaton is blocked in location `tpl_a_t_s2` until the callee (*ie.* the automaton that describes `tpl_activate_task` function) notifies the termination of the function by setting the shared variable `activate_task`.

V. Verification and certification

An OSEK/VDX certification consists in the execution of a test suite, i.e. a set of applications. Each test outcome may be a

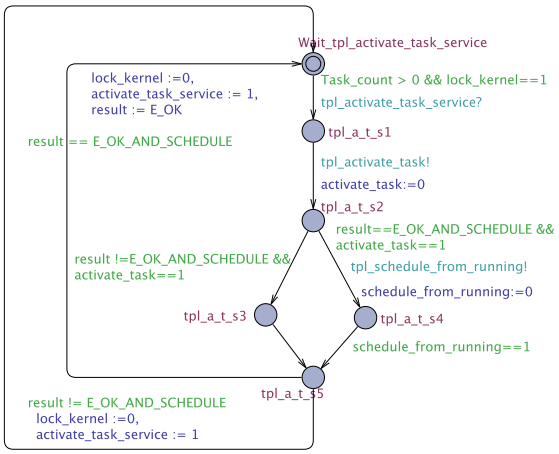

Fig. 3. model of tpl_activate_task_service function

success or a failure. The test suite specification is available on the OSEK/VDX portal. A first document [10] describes about 250 test cases. Each test case corresponds to a service call in a known state and its expected result. For instance test case 2 and test case 34 are as follow:

Case	Action	Expected result
2	Call ActivateTask() from non-preemptive task on *suspended* basic task	No preemption of *running* task. Activated task becomes *ready*. Service returns E_OK
34	Call Schedule() from task.	*Ready* task with highest priority is executed. Service returns E_OK

A second document [11] describes the test procedure: 37 test sequences that concatenate up to 20 test cases. Based on this specification it is possible to build a model of each test sequence and an observer that will allow to verify the properties corresponding to each test case in the sequence. For instance, test sequence 2 is an application with 3 tasks as modeled in figure 4 with $priority(t_3) > priority(t_2) > priority(t_1)$. The expected execution sequence is modeled by the observer shown in figure 5. If the expected sequence occurs, the observer reaches its final state. Then the following property is verified: $AF\ observer.success$ or "All paths lead fatally to state *success*".

All the test suite can be modeled by a combination of observers and test sequences models. On the complete operating system model, the full OSEK/VDX certification test suite may be applied. On a pruned model specialized for an application, all the test sequences corresponding to services used for this specialization can be checked to ensure that the specialization did not introduced unwanted behaviours, nor suppress expected ones.

VI. Conclusion

We have proposed a safer method for the generation of application-specific RTOS based on formal methods. A complete model of the operating system is built with finite automata extended with variables. Variables are used to model all RTOS object descriptors such as tasks, resources, alarms etc. RTOS service and functions are modeled by a combination of finite automata and imperative code. The application requirements are also modeled by a set of finite automata extended

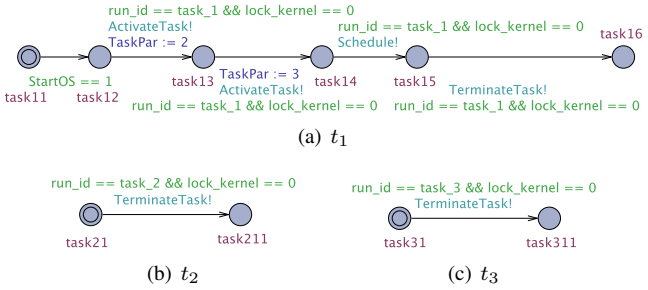

Fig. 4. Model of the tasks used in test sequence 2.

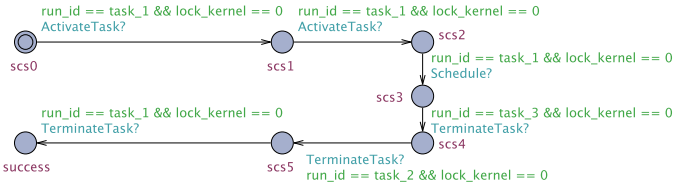

Fig. 5. Observer of the model.

with variables, and then combined with the RTOS model. The resulting model describes the application deployment on the RTOS. By using a reachability analysis, all unwanted behaviors are remove from this model. The resulting model contains only the paths that are traversed during the RTOS execution. Model-checking is used to formally verify that a set of properties describing the expected behavior of the RTOS hold in this model. Finally, the entire code of the corresponding RTOS is generated from the pruned model.

References

[1] T. E. Anderson, "The case for application-specific operating systems," in *Workshop on Workstation Operating Systems (WWOS)*, 1992.

[2] AUTOSAR GbR, "Specification of operating system," 2009.

[3] J.-L. Béchennec, M. Briday, S. Faucou, and Y. Trinquet, "Trampoline an open source implementation of the OSEK/VDX RTOS specification," in *IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2006.

[4] J. Bengtsson, K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi, "UPPAAL — a Tool Suite for Automatic Verification of Real–Time Systems," in *Workshop on Verification and Control of Hybrid Systems III*, ser. LNCS, no. 1066, 1995.

[5] C. Boke, M. Gotz, T. Heimfarth, D. El Kebbe, F. Rammig, and S. Rips, "(re-) configurable real-time operating systems and their applications," in *IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS)*, 2003, pp. 148–155.

[6] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski, "Frama-c," in *Software Engineering and Formal Methods*. Springer, 2012.

[7] P. Druschel, "Efficient support for incremental customization of os services," in *International Workshop on Object Orientation in Operating Systems*, 1993.

[8] G. Kiczales, M. Theimer, and B. Welch, "A new model of abstraction for operating system design," in *International Workshop on Object Orientation in Operating Systems*, 1992.

[9] D. Lohmann, W. Hofer, W. Schröder-Preikschat, J. Streicher, and O. Spinczyk, "Ciao: An aspect-oriented operating-system family for resource-constrained embedded systems." in *USENIX Annual Technical Conference*, 2009.

[10] OSEK Group, "OSEK/VDX OS test plan version 2.0," April 1999.

[11] ——, "OSEK/VDX OS test procedure version 2.0," April 1999.