Evaluation of Safety Rules in a Safety Kernel-Based Architecture

Eric Vial and António Casimiro

Universidade de Lisboa, Faculdade de Ciências
evial@lasige.di.fc.ul.pt, casim@di.fc.ul.pt

Abstract. Kernel-based architectures have been proposed as a possible solution to build safe cooperative systems with improved performance. These systems adjust their operation mode at run-time, depending on the actual quality of sensor data used in control loops and on the execution timeliness of relevant control functions. Sets of safety rules, defined at design-time, express the conditions concerning data quality and timeliness that need to be satisfied for the system to operate safely in each operation mode.

In this paper we propose a solution for practically expressing these safety rules at design-time, and for evaluating them at run-time. This evaluation is done using periodically collected information about safety-related variables. For expressing the rules we adopt the XML language. The run-time solution is based on a safety rules evaluation engine, which was designed for efficiency and scalability. We describe the architecture of the engine, the solution for structuring data in memory and the rule evaluation algorithm. A simple sensor-based control system is considered to exemplify how the safety rules are expressed.

1 Introduction

Safety is typically a fundamental concern when designing and developing vehicular autonomous systems like autonomous cars, airplanes or boats. System safety is usually proved at design-time, for which assumptions on system properties have to be made (e.g., fault and timeliness assumptions). These assumptions are required to hold with very high probability when developing safety-critical systems. A difficult problem when moving towards more complex systems performing more complex functions is that failure modes also become more complex and the system behavior tends to be less predictable. This is amplified when considering cooperative systems, which need to interact over wireless communication networks. In such systems, making assumptions on bounded message delays becomes a hard exercise: either these assumptions are pessimistic, leading to inefficient solutions, or additional resources must be used to improve the characteristics of communication subsystem, increasing the cost of the solution.

To address this problem, the solutions proposed in literature typically suggest to separate the system in different parts. The properties assumed for each part will be different, resulting in systems with a hybrid architecture. Notable examples include the Simplex model [4] and the Timely Computing Base model [5].

A. Bondavalli et al. (Eds.): SAFECOMP 2014 Workshops, LNCS 8696, pp. 27–35, 2014.

In both cases, part of the functions will be executed in a more predictable way (under a stronger failure or timeliness model), while other functions, the complex ones, will execute in a less predictable part of the system. The system separation in two parts makes it easier to enforce the properties assumed for the "better" part, while safety is ensured by making the system adjustable at run-time: the complex functions will only be used for control when a certain set of assumptions is satisfied. For that, the system must encompass a safety manager that observes relevant variables, verifies if predefined safety rules (assumptions) are met, and adjusts the system configuration and operation whenever necessary.

This idea was explored in the KARYON project, which defined a generic architectural pattern for the development of sensor-based autonomous and cooperative systems [2]. The KARYON architecture is based on a safety kernel that performs the mentioned observation of safety-related variables and determines the adequate operation mode. The safety kernel is also responsible to drive the necessary adjustments or reconfiguration actions, according to the operation mode that was determined as the safe one. There is a set of safety rules that has been defined at design-time for each mode of operation to perform the desired functions safely. They are used at run-time by the safety kernel, which periodically evaluates if they are being satisfied, given the collected safety information. KARYON also proposes solutions to abstract specific sensor faults, defining an abstract sensor model that allows sensor data to be characterized by a *validity* attribute [1]. Therefore, safety rules are expressed in terms of *validity requirements*, as well as in terms of *timeliness requirements*. We note that safety rules, once defined in design-time, and once being considered for safety analysis, will not change in run-time.

This paper focusses on the design and the implementation of an engine that performs the run-time verification of safety requirements expressed in safety rules. This engine is one of the main components the safety kernel and hence has to perform the verification efficiently. In addition, we also devised a solution that deals with scalability issues and may thus be useful for more realistic applications, involving a large number of safety rules. The paper explain how the safety rules can be expressed using the XML notation, how they are parsed and stored in memory and what is the algorithm performed by the safety manager engine to evaluate safety based on collected safety information.

The paper is structured as follows. In Section 2 we provide a brief overview of the safety kernel components. The design and the implementation of the relevant components for evaluating safety at run-time are presented in Section 3. An example case is considered in section4, to illustrate how safety rules are defined. Finally, Section 5 concludes the paper.

2 Definitions and Concepts

We consider a system in which several (possibly cooperative) functions can be executed. Nominal system components required for the execution of these functions include: *sensors*, *actuators*, *computation* and *communication* components. Each

of these components can be used to support multiple functions. Each function can be provided with several *levels of service* (LoS), depending on the components that are being used and/or the *performance level* of each component. Some components can exhibit uncertain timeliness, but some of them (used to execute the functions with a minimum guaranteed LoS) must always be timely.

Besides the nominal system components, the system includes a safety kernel that is responsible for adjusting the performance level of specific components or reconfiguring the system, such that each function will be executed with a desired level of service (LoS). The safety kernel is necessarily in the predictable part of the system. For its operation, the safety kernel collects timeliness and sensor data validity information. It then uses this information to verify if safety rules are satisfied, determining the adequate LoS for each function. Depending on the combination of LoS for the different functions, a specific system configuration and/or component performance level is enforced.

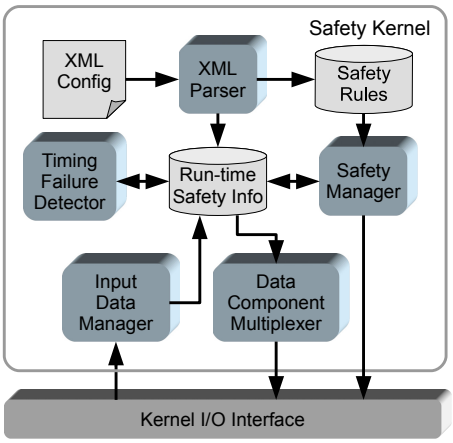

Fig. 1. Safety kernel components

Figure 1 gives an overview of the safety kernel components. At startup the *XML Parser* reads the local configuration, builds a *Safety Rules* repository and initializes *Run-time Safety Information* (RSI) structures. Therefore, the configuration file includes both safety rules and *unit* definitions. A unit corresponds to a safety kernel input (collected data), output (adjustment data – typically a component performance level) or locally calculated values (for instance, the acceptable LoS for some function). A safety rule is a boolean expression involving combinations of static values (bounds) and unit identifiers. A safety rule is meaningful for a specific LoS of some function. For instance, function F_2 can only be safely executed in LoS 1 when data validity V_0 is greater than 50 and V_1 greater than 70. This is expressed as:

$$F_2(LoS1) \rightarrow V_0 > 50 \wedge V_1 > 70$$

The *Input Data Manager* receives data inputs from the external (nominal system) components and updates the RSI. The *Timing Failure Detector* (TFD) is responsible for checking if certain data inputs have been received from external components within predefined temporal bounds. This TFD executes periodically, during each execution round of the safety kernel. When the TFD detects a timing failure (some data, which might be just an heartbeat, has not been timely

produced at the safety kernel interface), it stores this information in the RSI unit corresponding to the untimely data. The *Data Component Multiplexer* selects, from two or more data inputs (collected from nominal components), one that is forwarded to its output. This is useful, for instance, when the nominal has two components providing the same data (e.g., a front distance value), one providing data with high validity, but taking an uncertain amount of time to produce this data, and the other providing data with lower validity, but always in a timely way. The Data Component Multiplexer selects, among the two values, the better one, if timely produced, and the lower validity one, otherwise. Finally, the Safety Manager is the central component as it evaluates at run-time if *Safety Rules* are satisfied given the RSI data.

3 Design and Implementation

In this section, we start by describing how the *Run-time Safety Information* and the *Safety Rules* are represented in memory. Then we explain the solution for parsing and storing safety rules and, finally, we address the safety manager.

3.1 Data Structures

Data structures must be simple to provide code robustness, but they are designed as well with the aim of reducing the computation time during the rule evaluation phase. The Run-time Safety Information (RSI) repository is initialized during system bootstrap and is updated at run-time with collected safety-related information. The RSI size depends on the number of units (inputs, outputs and internal variables) declared in the configuration file. As this size is not changed at run-time, we use a single dimension array to store the units. Each unit structure contains several fields, including a pointer to related safety rules, which set requirements on this unit, a timeliness status, which may be relevant for units with timeliness constraints, a data validity value, a level value that may be used to store performance levels or levels of service (this is clarified ahead in the text), and some other attributes.

The safety rules are also built at bootstrap from the configuration file. We note that one possible design approach would be to simply hard code the safety rules within the safety kernel, thus avoiding the need for specifying them in a configuration file, and consequently processing them at bootstrap. However, we decided to follow an approach that provides some additional flexibility and leads to a generic safety kernel implementation. Safety rules can be updated without the need for recompiling the code and loading it on the board, which is particularly advantageous during the development process. And the safety kernel core is totally independent of the specific application, which can facilitate verification and validation activities.

Given that safety rules need to be checked in every execution cycle, within a limited amount of time, a fundamental requirement is to devise a solution for storing them in memory, such that safety management is efficient and scalable.

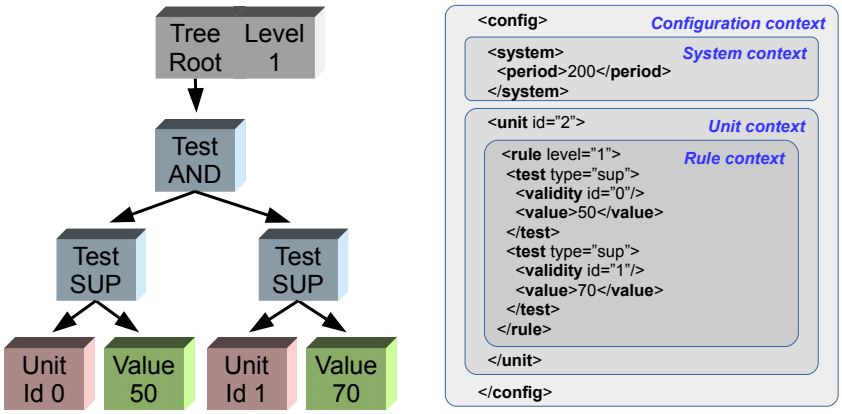

Fig. 2. Basic safety rule definition

This is particularly necessary if considering that in real systems the number of safety rules will tend to be very high. The concrete number will depend on the amount of functionalities that may be performed by an autonomous vehicle, on the number of system variables that may have to be checked in run-time, and on the number of levels of service considered for each functionality. We addressed this requirement by adopting a tree-based data organization, where the root node for each safety rule contains the associated level of service (LoS) and a pointer to the top child node of the tree. This tree is created during the XML Parsing. This kind of structure allows for efficient rule parsing at run-time, using the algorithm described in Section 3.3.

The tree corresponding to the basic rule example from Section 2 is shown in Figure 2 (left). Three different types of nodes can be used in the tree: test nodes, unit id nodes and value nodes. Test nodes store boolean operations, like *AND*, *OR*, *EQUAL*, *DIFF* or *SUP*, among others. Each unit id node contains the index of a unit in the RSI array. According to the way a unit id is defined in the configuration, it contains either a data validity value or a level of service/performance level value. In the example, the two units (ids 0 and 1) will contain data validity values. Different rule trees can refer to a single unit when there are multiple constrains (safety rules) related to a certain safety-related variable. When this happens, the several rule trees are level-sorted (from the higher to the lower level, as defined in the root node) in a linked list to which the unit will point. Finally, value nodes contain constants (bounds) against which the unit values will be checked.

3.2 XML Parsing

A lot of *XML Parsers* are described in the literature and many of them are available for free. These parsers usually offer a large range of functionalities and

may be not portable to RTEMS environments. As a consequence, we chose to develop our light XML Parser with only some basic features. Its architecture is similar to the open markup parser available in Glib library [3].

The XML Parser is a simple context-based parser. In each context there are two callback functions that are used for opening and closing markup tags. During the parsing, these functions are called to initialize the RSI array and the safety rule trees. Context switching is performed inside the callback function according to the parsed XML tag.

Figure 2 (right) shows an XML configuration file implementing the basic safety rule example given in Section 2. A configuration file admits four context levels. The *configuration context* is the default one, while the *system context* is used to initialize global system parameters. For instance, in this example the safety kernel period is set to 200 ms. The purpose of the *unit context* is to define a new unit. Finally, the *rule context* is used to build a safety rule tree associated to a given unit. Note that besides the output unit with id 2 (which allows to set the performance level of some application component), two additional units are created (ids 0 and 1) to store data validity values. The output unit will hold the value 1 when the (only) rule evaluates to true, and 0 otherwise. A node stack allows to internally store the nodes and assemble the tree.

3.3 Safety Rules Evaluation

At run-time the Safety Manager will periodically scan the RSI array. For each unit with at least one defined rule (some units, like units 0 and 1 from the example, do not have any associated rule), the Safety Manager evaluates them starting with the rule with the highest level. The rationale is to first evaluate if the conditions to perform some function at the highest level of service are satisfied. When they are not, then other safety rules will be checked. Therefore, the evaluation stops when a rule is satisfied or when the end of the rule list is reached. In the latter case, this means that the function has to be executed at the lowest LoS (level 0). At the end of the process, the Safety Manager updates the level field of internal units (those holding the acceptable LoS for some function) and of output units (holding the performance level level of specific components). The rule evaluation functions are the following:

```
 1: function LEVEL(rule_list)
 2:     for all rule ∈ rule_list do
 3:         node_list ← rule.root
 4:         if AND(node_list) then
 5:             return rule.level
 6:         end if
 7:     end for
 8:     return 0
 9: end function
10:
11: function AND(node_list)
12:     for all node ∈ node_list do
13:         if ¬EVAL(node) then
14:             return false
15:         end if
16:     end for
17:     return true
18: end function
19:
20: function EVAL(node)
21:     switch node.type do
22:         case test
23:             return TEST(node)
24:         end case
```

```
25:         case unit
26:             return UNIT(node)
27:         end case
28:         case value
29:             return true
30:         end case
31:     end switch
32: end function
33:
34: function TEST(node)
35:     node_list ← node.test.childs
36:     switch node.test.type do
37:         case sup
38:             if ¬AND(node_list) then
39:                 return false
40:             end if
41:             return COMPARE(node) > 0
42:         end case
43:         ...
44:     end switch
45: end function
46:
47: function UNIT(node)
48:     id ← node.unit.id
49:     unit ← unit_array[id]
50:     return unit.status
51: end function
```

The *level* function (line 1) evaluates the unit rule list. The *and* function is first called, as the top-level node is always an *AND* in any rule tree. This first node gathers all conditions required for the rule to be satisfied. The *eval* function (line 20) evaluates a node according to its type. In the *test* function (line 34) we only show the SUP operator (line 37). First we check the timeliness status of both operands by recursively calling the *and* function. If the evaluation returns true, we compare the values of both operands (line 41). The *unit* function (line 47) is called to evaluate a unit and returns its timeliness status.

4 Example Application

We consider an example application in which two cooperative functions, CF_A and CF_B, are implemented. These functions use two sensors, *S1* and *S2*, and five functional components, from *C1* to *C5*. Both sensors provide a data validity value associated to the sensor data they produce, which is sent to the safety kernel (*V1* and *V2*). *C4* is a multi-component with two implementations, *C4'* and *C4"*, corresponding, respectively, to performance levels *PL1* and *PL0*. According to the execution timeliness of *C4'*, called ET_{C4-PL1}, the *Data Component Multiplexer* will forward the selected value from *C4* to *C5*. Finally, *C1* is a component below the hybridization line able to execute with three different performance levels (from PL2 to PL0). We also consider that the safety rules for both functions are the following (the bounds have to be defined at design-time, and it must be proven that the functions will be safely performed in each LoS when the safety rules are met):

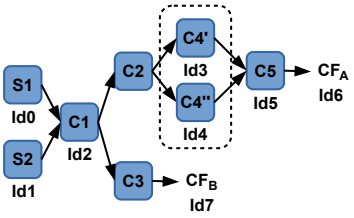

Fig. 3. Example functions

$CF_A\_LoS(LoS3) \rightarrow V1 > 80 \wedge ET_{C4-PL1} < D_{C4}$
$CF_A\_LoS(LoS2) \rightarrow V1 > 60 \wedge ET_{C4-PL1} < D_{C4}$
$CF_A\_LoS(LoS1) \rightarrow V1 > 60$
$CF_A\_LoS(LoS0), otherwise$

$CF_B\_LoS(LoS3) \rightarrow V1 > 80 \wedge V2 > 70$
$CF_B\_LoS(LoS2) \rightarrow V1 > 80$
$CF_B\_LoS(LoS1) \rightarrow V1 > 60$
$CF_B\_LoS(LoS0), otherwise$

The next table shows the performance levels of *C1* and *C4* in dependence of the *LoS* of both functions. All invalid combinations have been removed. On the right side of the table, we provide a possible set of expressions that can be used to calculate these performance levels.

CF_A	CF_B	$C1$	$C4$
LoS3	LoS3	PL2	PL1
LoS1	LoS3	PL2	PL0
LoS3	LoS2	PL1	PL1
LoS2	LoS1	PL1	PL1
LoS1	LoS1	PL1	PL0
LoS0	LoS0	PL0	PL0

$C1\_PL(PL2) \rightarrow CF_B\_LoS = 3$
$C1\_PL(PL1) \rightarrow CF_B\_LoS > 0$
$C1\_PL(PL0), otherwise$

Given all the above expressions, required to determine the feasible LoS for each function and the corresponding component performance levels, it is possible to create an XML configuration file. An identifier must be first assigned to each unit (e.g., ID0 for *S1*, ID1 for *S2*, ...), and this allows the proper references to be made in the configuration file. Note that there are no IDs for *C2* ad *C3* as they are not involved in any expressions. A subset of the resulting configuration file is presented below.

```
<?xml version="1.0"?>
<config>
    <!-- C1 component -->
    <unit id="2">
        <mode>update</mode>
        <rule level="2">
            <test type="equal">
                <level id="7"/>
                <value>3</value>
            </test>
        </rule>
        <rule level="1">
            <test type="sup">
                <level id="7"/>
                <value>0</value>
            </test>
        </rule>
    </unit>
    ...
    <!-- Function A -->
    <unit id="6">
        <rule level="3">
            <test type="sup">
                <validity id="0"/>
                <value>80</value>
            </test>
            <test type="equal">
                <level id="5"/>
                <value>1</value>
            </test>
        </rule>
        <rule level="2">
            <test type="sup">
                <validity id="0"/>
                <value>60</value>
            </test>
            <test type="equal">
                <level id="5"/>
                <value>1</value>
            </test>
        </rule>
        <rule level="1">
            <test type="sup">
                <validity id="0"/>
                <value>60</value>
            </test>
        </rule>
    </unit>
    <!-- Function B -->
    <unit id="7">
        ...
    </unit>
</config>
```

5 Conclusion

This paper describes the solutions developed in the KARYON project for specifying safety rules in configuration files, for storing safety rules and safety data in memory, and for evaluating safety at run-time. They were designed with the objective of being simple but effective, addressing performance and scalability requirements. This simplicity facilitates the calculation of upper bounds for safety rule evaluation time. These solutions have been implemented and are being used in the KARYON vehicular demonstration prototypes.

Acknowledgements. This work was partially supported by the EU's FP7 through project KARYON, under grant agreement No. 288195, and by the FCT, through the Multiannual program.

References

1. Brade, T., Zug, S., Kaiser, J.: Validity-based failure algebra for distributed sensor systems. In: SRDS, pp. 143–152 (2013)
2. Casimiro, A., Kaiser, J., Schiller, E.M., Costa, P., Parizi, J., Johansson, R., Librino, R.: The karyon project: Predictable and safe coordination in cooperative vehicular systems. In: 2013 43rd Annual IEEE/IFIP Conference on Dependable Systems and Networks Workshop (DSN-W), pp. 1–12. IEEE (2013)
3. GLib Project: Rsimple xml subset parser, version 2.37 (2014)
4. Sha, L.: Using simplicity to control complexity. IEEE Software 18(4), 20–28 (2001)
5. Verissimo, P., Casimiro, A.: The timely computing base model and architecture. IEEE Transactions on Computers 51(8), 916–930 (2002)