Hazardous Behavior Analysis for Complex Pre-Existing Software in Safety-Critical Context: Approach and Application to the Linux Dynamic Memory Allocator

Raffaele Giannessi, Fabrizio Tronci, and Alessandro Biondi, *Member, IEEE*

***Abstract*—As system designers are transitioning to the usage of pre-existing software architectural elements to reduce time-to-market and costs, they face challenges in safety-critical applications. Pre-existing software may have been implemented without following any safety and/or quality standard, and its documentation may be incomplete or unclear. As recommended by part 6 of the functional safety standard ISO 26262 (product development at the software level), stringent rules and guidelines must be followed during the implementation of a new software. The qualification of a pre-existing software according to ISO 26262 may hence be very time consuming and expensive if not addressed in a structured way. This work presents *STPA for Pre-existing Software* (STPA-PES), a new software hazardous behavior analysis approach designed to identify criticalities or abnormal conditions in complex pre-existing software to be mitigated with adequate safety measures. The major challenge in such a process consists in the definition of a model that correctly represents the system under analysis when the architectural design of pre-existing software is not available. As a relevant example, the proposed approach is finally applied to dynamic memory allocator (DynMA) in Linux kernel to show how STPA-PES is able to derive safety requirements of this software architectural element.**

***Index Terms*—System design and analysis, Software and System Safety, Operating Systems, Systems and Software, Reusable Software, Control structures**

I. Introduction

MANY safety standards have been defined over the years to drive software suitability for safety-critical purposes. Their goal is typically to regulate the safety life cycle and functional safety aspects of products in different fields. A relevant example is the ISO 26262 [1], the international functional safety standard for addressing field-specific needs of electrical and/or electronic (E/E) systems within road vehicles. The recommendation provided by safety standards is to apply safety guidelines with a stepwise approach during the product lifecycle, thus reducing the presence of critical issues in the final product.

Software reuse is increasingly becoming common practice for designing safety-critical systems to reduce development time and cost. However, the integration of pre-existing software for safety-critical purposes should be carefully controlled, as it may have been implemented without following any safety standard. Its analysis for suitability for usage in safety-critical applications may be complex and time-consuming, especially in the case of software systems with a large code base such as an operating system (OS). In this context, several companies are currently working together on international projects such as ELISA [2], which has the goal of defining strategies and implementing tools to qualify Linux-based safety-critical applications according to ISO 26262. The challenges of such projects are many also because, as pointed out by Israeli and Feitelson [3], Linux is an open-source operating system in continuous evolution. Moreover, it has a huge number of possible configurations that contribute in exacerbating its complexity.

System design weaknesses or criticalities can be found by applying different failure analysis techniques [4], which typically represent the system as a combination of events or controllers. One of the most recent approaches is *system-theoretic process analysis* (STPA) [5], a control-oriented technique aimed at identifying hazards of complex systems using a systematic process based on the interactions between system controllers. STPA is particularly suitable for driving the design of control systems. However, it was not conceived to be applied on software architectural designs, and, in particular, its original definition does not fit with middleware structure or low-level software whose design is not control-oriented.

Contribution. This paper proposes *STPA for pre-existing software* (STPA-PES), a methodology intended to drive the generation of safety requirements starting from hazardous behavior analysis of complex pre-existing software. STPA-PES extends the STPA to enable the analysis of generic software systems (i.e., not only control systems), particularly focusing on software abstraction modeling. The produced software model aims at representing the system without oversimplifications or over-complexities. As a case study, STPA-PES has been applied to the Linux Kernel dynamic memory allocator (DynMA)—a complex low-level architectural element that may compromise the integrity of the intended functionality in safety-related applications. The choice of Linux DynMA is due to the fact that many safety standards and software guidelines suggest to avoid the usage of such a feature, thus requesting

R. Giannessi is with Evidence S.R.L., Pisa 56124, Italy and also with the Scuola Superiore Sant'Anna, Pisa 56124, Italy.
E-mail: raffaele.giannessi@virgilio.it
F. Tronci is with Evidence S.R.L., Pisa 56124, Italy.
E-mail: fabrizio.tronci@hotmail.it
A. Biondi is with the Scuola Superiore Sant'Anna, Pisa 56124, Italy.
E-mail: alessandro.biondi@santannapisa.it

TABLE I
RELATED WORK COMPARISON

Paper	Applicable scope	Approach	Referenced in ISO 26262-9.8
Vesely et al. [6]	HW	Top-down	Yes (FTA)
Towhidnejad et al. [7]	SW	Top-down	Yes (FTA)
Reifer [8]	SW and HW	Bottom-up	Yes (FMEA)
Takahashi et al. [9]	SW and HW	Hybrid	Yes (FTA and FMEA)
Leveson et al. [10]	SW and HW	Bottom-up or top-down	No
Cherfi et al. [11]	SW and HW	Top-down	Yes (Markov models)
Papazoglou [12]	SW and HW	Top-down	Yes (ETA)
Čepin [13]	Component-based systems	Bottom-up	Yes (Reliability block diagram)
Lawley [14]	Control systems	Top-down	Yes (HAZOP)
Leveson [15]	Control systems	Top-down	No
Leveson and Thomas [5]	Control systems	Top-down	No
Friedberg et al. [16]	Security-based systems	Top-down	No
Yu et al. [17]	Security-based systems	Top-down	No
Abdulkhaleq et al. [18]	SW-intensive systems	Top-down	No
This paper	**Pre-existing SW**	**Hybrid**	**No**

adequate software measures to allow its usage in safety related products. An example can be found in ISO 26262-6 where the dynamic allocation of objects and variables is allowed only in the presence of a safety mechanism responsible for minimizing possible faults.

Paper structure. The rest of the paper is organized as follows. Section II reports current approaches to address functional safety in software, highlighting the uncovered challenges that our work expects to overcome, while Section II-A briefly recaps the STPA technique. Section III presents the STPA-PES. Section IV shows how to apply it, step by step, to Linux DynMA. The results are then reported in Section V. Section VI concludes the paper and discusses future work.

II. RELATED WORK

Starting from some classic failure analysis methods, this section reviews techniques available in the literature to address safety issues in software systems.

One of the most famous approaches is fault tree analysis (FTA) [6], which is a top-down technique that uses boolean logic to represent the system failures by representing each unit as a boolean operator. The approach was originally used to model hardware failures and has later been adapted by Towhidnejad et al. [7] for software design. While being particularly useful during the software development phase, it is not suitable for the analysis of pre-existing software, since it requires a deep knowledge of the architecture under analysis. Indeed, in the case of complex pre-existing architectural components, the integrator might not fully understand the design of the components by relying on their source code only. Failure modes and effect analysis (FMEA) [8] is a bottom-up technique that aims at identifying potential causes of failure inside each component and the related produced effect. Takahashi et al. [9] combined FTA with FMEA to improve the safety level of control software. Leveson et al. [10] described a way to model safety-critical, real-time systems failures and faults by using Petri nets for their design and analysis. Such a technique is useful to represent the system by also taking into account timing event relations, but the resulting model is hard to be analyzed, especially for software with a large code-base. Cherfi et al. [11] presented an approach based on Markov chains to model the behavior of E/E systems to fill uncovered gaps of ISO 26262. The goal of event tree analysis (ETA) [12] is instead to find chains of events that may move the system to a fault. Such techniques are referenced in ISO 26262 as analysis methods to address random hardware failures, but they are not suitable to be applied to software.

With *reliability block diagrams* [13] the system is represented as a composition of components placed in series or parallel. Such a model allows the analysis of the probability of system failure, but it does not apply to large monolithic software. A systematic way to analyze the entire system is the hazards and operability analysis (HAZOP) [14]. With this approach, the overall design is decomposed into simpler units to be reviewed separately. System-theoretic accident model and processes (STAMP) [15] proposes a model about how accidents occur, which put the basis to define the STPA, suitable to analyze very complex systems. However, the first steps are not guided systematically, thus basing the analysis on the analyzer's capacity to manage trade-offs. Leveson [19] better analyzes the cause of flaws identified during the STPA, highlighting that software is one of them.

During the years, STPA has been used as a base to build suitable methods for different applications. An example is STPA-SafeSec [16], which has been proposed as a methodology to analyze the system from both safety and security points of view. Another example is STPA-DFSec [17], an approach based on a data-flow structure that considers each component as a composition of a set of functions. However, these methods

focus on both system-level safety and security during the development phase, and do not hence trivially transfer to the analysis of pre-existing, general-purpose software.

Table I reports a comparison of the related work, where each paper is classified depending on: **(i)** the scope for which the method is applicable (i.e., SW, HW and control system), **(ii)** the strategy followed by the method (top-down, bottom-up or hybrid), and **(iii)** if it is referenced or not in ISO 26262-9.8.

Abdulkhaleq et al. [20] focused on the application of STPA to systems with high complexity. In their work, the STPA is applied at different levels such as the vehicular, the system, and finally the component levels. Such an approach introduces a systematic way to represent the system control structure in a hierarchical manner, but it analyzes again the software as a whole. STPA-based approach for software-intensive systems (STPA-SwISS) [18] extended the STPA to enable its application for software. The approach can be applied either during the development process or to pre-existing software. However, it is hard to apply it to complex pre-existing software like an entire OS. Rejzek et al. [21] managed high-complexity systems by representing the hierarchical control structure in different views but, as the STPA, their approach applies to control-based systems instead of software in general. The system analyzer freedom is confined in the work proposed by Zhong et al. [22], where UML views are used to enforce the STPA framework. Abdulkhaleq et al. [23] proposed a safety engineering approach based on STPA to develop safe software. Both are interesting approaches that enable engineers to develop safe software, but their application to a pre-existing general-purpose software is not straightforward.

Thieme et al. [24] presented a way to decompose a software system into its functions. However, the level of decomposition should be carefully selected to extrapolate a model that represents sufficiently the software under analysis without providing more details than needed.

A. Background on STPA

This section introduces the basics of STPA, on which the approach proposed in this work is based. As described by the STPA guidelines [5], STPA is composed of four phases.

The first one involves the definition of the purpose of the analysis, identifying system boundaries, losses, and hazards. With loss the STPA defines something that must be prevented, like a mishap or an adverse event, while an hazard is defined as the system state or a set of conditions that may lead to a loss. In this phase, each hazard should be analyzed to find the worst-case conditions that may produce the identified losses.

In the second phase, the analysis builds the system control structure. Such a model represents the system as a group of controllers interacting with each other by sending control actions and feedback. The STPA handbook [5] highlights the importance of this phase by reporting some guidelines and examples to prevent common mistakes. In particular, important details should not be omitted if the system is already implemented, but the final model complexity should be manageable by the next steps of the analysis.

The control structure is taken as input for the third phase, which consists of the identification of the *unsafe control actions* (UCAs). In this phase, each control action in the control structure should be analyzed in order to define how and when it may lead to a hazard. Not providing, providing with the wrong parameter, and providing in the wrong order the control actions are classical examples of conditions that characterize a UCA. The identified UCAs should finally be written in the cells of a table that reports the control actions in the rows and the type of UCA (not provided, provided, too early to late or out of order and stopped too soon or applied too long) in the columns.

The last phase of the STPA consists of the identification of the loss scenarios, which represent the situations where a control action may lead the system to a hazard. This phase consists of two sub-phases. The first one drives the generation of a table that maps each UCA to the related loss scenario, which has to describe, by means of natural language, how the UCA may produce the related hazard. The second one identifies scenarios that lead control actions to be improperly executed or not executed at all.

III. Proposed approach

A. Motivation

STPA is a very powerful technique that enables software hazardous behavior analysis of complex systems. Its application mainly involves control-based products and is typically applied at the system level. This is highlighted by the second phase of STPA, where the model to be built is a control structure. Such a model describes controllers and controlled processes as the basic elements composing the entire system. The approach presented in this work instead intends to be used also on low-level embedded software, which is not straightforward to be treated as a composition of controllers. Moreover, in classic STPA loss scenarios are typically expressed in natural language, which forces the analyzer to interpret them in order to produce tests and requirements.

The hierarchical control structure built during the second phase of STPA may describe the model of complex systems. However, the controllers granularity is completely managed by the system analyzer, thus producing a model that represents the system in a way that is not unique.

In order to overcome such limitations of STPA, this paper proposes different improvements:

- Huge or useless control structure will be avoided by moving the last phases of STPA to the component level instead of considering the entire system as a whole.
- Pre-existing software control structure extrapolation will be guided systematically.
- Loss scenarios interpretation is enforced to remove ambiguities and drive the requirements generation phase.

B. STPA-PES approach in detail

The analysis of complex pre-existing software may be very challenging and time-consuming if not systematically addressed. System decomposition [25] is a helpful approach that may produce simpler sub-systems that can be analyzed in parallel. This work proposes to first, apply STPA SwISS at the

system level. Then, each complex software component should be analyzed by splitting STPA into two parts as reported by Figure 1: *global* and *for each safety-related (SR) feature*.

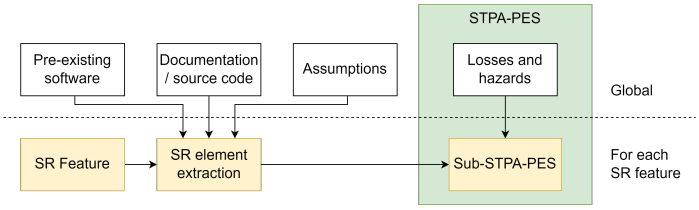

Fig. 1. Once applied STPA-SwISS at system level STPA-PES can be applied to each software controller. The Figure shows the proposed approach based on STPA: phase 1 is applied at system level, while other phases are applied to each SR feature.

In particular, the identification of losses and hazards may follow the process as described in the STPA handbook [5], thus referring to the entire pre-existing software architectural element. Instead, the next steps must be driven for each decomposed software element, defined here as the block that is responsible to provide a certain SR feature. A control structure representing a complex system as a whole may be too intricate and so error-prone. The result is that the next phases of the analysis will produce very abstract UCAs and loss scenarios, which may not help in finding weaknesses in the system. The proposal is instead to construct a decomposed control structure by separating the set of SR features from non-SR (NSR) ones. The separation between SR and NSR features can be done either through a dependent failure analysis or by considering all the features as SR. The analysis of each software feature can be guided by assuming the rest of the system safe and correctly used. The rationale for this reasoning is that verifying the integrity of each SR feature is in the scope of the corresponding safety analysis, while the dependencies among different SR features are verified with dynamic and static tests of their interfaces. Under these assumptions, the STPA-PES can move on with next steps.

The *SR element extraction* is the phase that enables the analysis of simpler and smaller blocks of code. As reported in Figure 2, such a procedure starts with the identification of functions in scope, which are functions that are usually called by functionalities that belongs to systems different than the one under analysis, asking for a specific service through an application programming interface (API).

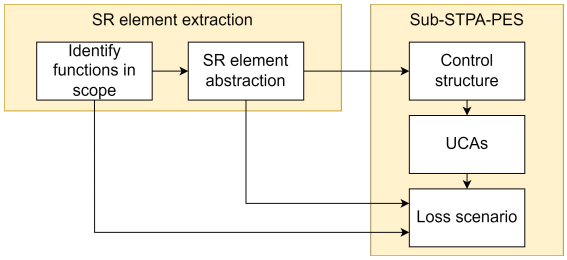

Fig. 2. Sub-STPA-PES phase in details: STPA phases 2 to 4 are applied here to an abstracted software component that represents a SR element providing a SR feature.

The *SR element abstraction* phase guides the analyzer in looking inside the SR feature under analysis. Without this phase, each software element would be treated as a black box, thus producing a control structure so general that the next phase of the STPA-PES will end up with very abstract loss scenarios.

C. Software element Sub-STPA-PES

1) SR element extraction and control structure modeling: The main contribution of this work can be found in the extension of the second phase of the STPA. In the STPA handbook [5], the detail and the rules about how to define the control structure are defined with a certain margin of freedom that is left to the system designer. STPA-SafeSec [16] tries to restrict this freedom by introducing the concept of system states, which, for general-purpose software, can be considered as the combination of *global and local contexts*. The first one is represented by the global variables and flags values, while the second one contains local variables, the stack, and the function currently executed. This paper proposes a complementary approach for pre-existing software architectural elements (that are not control-oriented) aimed at identifying controllers in the control structure.

Figure 3 highlights the main steps to drive the *SR element abstraction* phase.

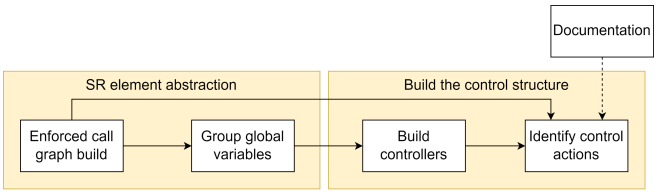

Fig. 3. SR element abstraction and control structure phases in details: before building the control structure the code is analyzed to identify controllers and their interactions.

It is based on the analysis of the global context (global variables and data structures) accessed by C functions at code level inside the enforced call graph, which reports dependencies among different subroutines and the global variables accessed by each function. The proposal is to group together global variables accessed by the same group of functions. The rationale is that each controller should be responsible to manage and update its own group of global variables. A similar approach has been proposed by Tyszberowicz et al. [25], where the authors applied a decomposition to a monolithic system to represent it as a composition of microservices. With this approach, the creation of a control structure starts with the identification of such variables and moves to the identification of functions that are used as interface to manipulate them. The controller body will be composed of groups of function calls hidden internally to avoid over-complexities in the model. At this stage, the controllers have been created and their interface defined, but no interaction has been modeled until now. Control actions among different controllers are modeled in the control structure by taking in consideration once again the enforced call graph. In particular, a control action should be added for each function call made from

one controller to another through its interface. Feedbacks instead, are identified by taking into account their respective return values. In classic STPA, parameters are usually hidden in control actions, thus producing ambiguities that may be misinterpreted in subsequent phases. The reuse of parameters found in functions will cause a mismatch in the level of abstraction currently used to represent the system. This work proposes to abstract parameters in control actions by exploiting the system knowledge obtained from the documentation of the system.

The abstracted model represents at this stage the infrastructure of the system, while dynamic aspects such as the functionality provided by a controller in response to a control action are still not defined. This work proposes to obtain and abstract such information by leveraging the documentation of related functions.

2) Unsafe control actions: Once defined the abstracted SR software element control structure, the next step is the identification of UCAs. In the STPA handbook [5] a UCA is defined as a control action that, in a particular context and worst-case environment, will lead to a hazard. The STPA handbook systematically drives this phase to fill a table that collects all the identified UCAs. The whole table can be reused here except for the "stopped too soon or applied for too long" column, which is not applicable to software modules that are not control-oriented.

In the context of software, UCAs can be considered as *actions* (either functions or low level actions like a read or a write at some specific memory location) that are either:

- executed in the wrong order;
- not executed when needed; or
- executed with wrong values as parameters.

Such UCAs may produce additional events or chains of function calls that will end up with an incorrect update of a global variable, an action that may break the abstracted software element state. Other consequences that a UCA may produce are a wrong control action sent to other controllers or wrong feedback.

3) Loss scenario: The identification of loss scenarios is the last phase of the STPA-PES and it takes the fourth phase of the STPA as a base. Here, the aim of this work is to identify scenarios potentially violating the original safety requirements allocated to the system under analysis, but also cascading failures that may compromise the freedom from interference (FFI) of software elements with different integrity coexisting in the same system. In ISO 26262-1 [1], FFI is defined as the absence of cascading failures among different elements in the system that could lead to a violation of a safety requirement.

The loss table must be generated by thinking about possible conditions and actions that may put the system in dangerous situations. The proposal in this case is to define each scenario as written in the STPA handbook, reporting also a realistic use-case at code level that may produce it. The STPA handbook splits this process in two sub-phases: the identification of scenarios that leads to UCAs and the identification of scenarios for improperly or not executed control actions.

Starting from the first one, such scenarios should be identified by considering possible combinations of events that lead the system to harmful conditions. At this stage, the parameters abstracted in control actions should be coupled to C function parameters to produce concrete loss scenarios. In an embedded software scenario suitable to the STPA-PES approach, events that may lead to UCAs should be identified by considering function calls and low-level actions like read or write to a certain memory location. A first example can be found in a wrong or unintended controller interface usage. In this context, such events are function calls not executed when needed, activated more times than expected or executed with wrong parameters. In this stage, also cascading failures can come out by analyzing the dependencies among different software element. Scenarios identified until here cover the first two columns of the UCAs table. Instead, UCAs in the third column are mapped as combinations of function calls, read or write actions.

A different reasoning should be done for the identification of scenarios for improperly or not executed control actions. While in physical systems controllers may be flawed or subject to failure, the software may contain weaknesses resulting in control actions that are not or improperly executed. By considering lower levels of the SR feature under analysis (for instance, the hardware level or other SR software elements) as correct, such a condition may produce either wrong feedback in the control action or a wrong data structure update. In every case, the condition to be controlled is very general, and the analyzer may not be able to generate the requirement to cover it. A control action that is not or improperly executed may also be generated due to a corrupted global context. However, these conditions should be avoided by protecting the memory data structure from other parts of the system and by testing and verifying the code in order to remove possible bugs internal to each controller. For this reason, such scenarios for control actions improperly executed are not considered in STPA-PES. Instead, not executed control actions may be produced for instance by resource contention. The proposal is to add one loss scenario for each control action present in the control structure in order to capture the hazard that may produce its missed execution.

This phase may also be helpful to generate some tests for the later verification of the effectiveness of mitigation strategies resulting from the analysis to control the weaknesses. For this purpose, a third column "code example" has been added in the loss scenarios sheet. It explains the scenario that should be managed with some code instructions, thus reducing possible incomprehension produced by scenarios described through natural language.

The sheet produced by the fourth phase lists possible scenarios that may lead to hazards, scenarios that should be managed by introducing one or more safety requirements. In particular, each scenario should be covered by at least one requirement. However, the procedure to derive safety requirement from loss scenario is case-dependent, and for this reason cannot be generalized.

D. Proposed approach in summary

The proposed approach can be summarized in the following operations.

Step0 - STPA at system level: STPA-SwISS should be applied first at the system level. In this step, each software controller is seen as a single component as a black box. Each complex software controller has to be analyzed by following the next steps.

Step1 - Documentation analysis: for a complex pre-existing software this is a crucial phase since it helps the analyzer in the comprehension of the code. The documentation is leveraged to understand the general behavior of the code under analysis and to spot, if present, a first model of the software. This phase provides help for step3 and step4.

Step2 - Identify losses and hazards: this phase coincides with the first phase of the STPA and it identifies critical conditions that the system must avoid. In particular, this phase produces the lists of losses and hazards as output to be used for next steps.

Step3 - SR element extraction: with this phase the analysis moves from the entire pre-existing software to SR elements inside. In particular, all the features provided by the software must be identified here. The documentation analyzed in step1 may help here in the identification of APIs related to each feature. For each API of a certain feature, static code analysis techniques should be leveraged to report the relative call graph, enforced to display also accesses to global variables.

Step4 - Build the control structure: once collected all the information, the analyzer must exploit the output of step1 and step3 to extrapolate a control structure abstracting the element that provides a SR feature. Each enforced call graph is abstracted here and merged to generate a structure that represent each SR feature provided by the pre-existing software. The behavior of each controller in such a structure is identified by exploiting the documentation. The final result of this phase is the control structure.

Step5 - Identify unsafe control actions: for each control action in the control structure built in step4 the analyzer must identify UCAs. This phase systematically generates a table that links each UCA to one or more hazard identified in step2.

Step6 - Identify loss scenarios: the final step is to identify loss scenarios that leads to UCAs or to control actions incorrectly or not executed. The UCAs table generated in step5 is the input for this phase together with hazards identified in step2. This phase generates the loss table, which links each UCA or control action incorrectly or not executed to the corresponding loss scenario that may produce it.

IV. Illustrative Example: STPA-PES applied to Linux dynamic memory allocation

Linux is an open-source complex monolithic kernel for general-purpose applications. Its large code base makes Linux a very good case to apply the proposed approach as an example. The following Guideline for applying the proposed approach to other software components are then reported in Section III-D.

A. Step0 - STPA SwISS applied to ELISA telltale example

In this work, we analyzed the ELISA telltale use-case [26], an example of an automotive scenario where the main application runs upon Linux. It focuses on the warning messages displayed in the Dashboard of the vehicle and the main actors in this scenario are the Safety App, the Watchdog, and the Safety signal source. The Safety signal source periodically sends signals to the Safety App. The Safety App is in charge of checking the signals received by the Safety signal source and resetting the watchdog if everything goes well. Whenever a signal is not received in time or is corrupted, the Safety App does not reset the watchdog and the safe state is triggered.

The identified losses are general and suitable for the whole system:

- **L-1**: Loss of life or injury to people.
- **L-2**: Loss of or damage to vehicle.
- **L-3**: Loss of or damage to objects outside the vehicle.

The identified hazards are instead:

- **H-1**: Driver is not informed about a system condition, leading to a collision or other harmful event.
- **H-2**: Driver is distracted by the display, leading to a collision or other harmful event.
- **H-3**: Driver becomes desensitized to a warning because it is incorrectly repeated, leading to a collision or other harmful event.
- **H-4**: Content on display compromises the driver (e.g. flickering image triggers epileptic fit), leading to a collision or other harmful event.

Every hazard may lead to either L-1, L-2 or L-3 under different conditions.

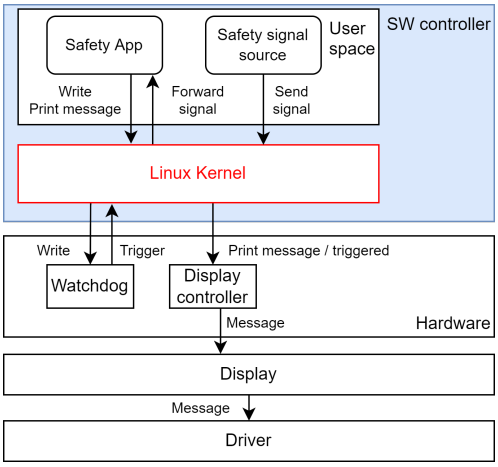

Fig. 4. Control structure representing ELISA Telltale example. Linux kernel is considered at this stage as a single component.

The control structure of the telltale use-case is reported in Figure 4. The main software components involved in this scenario are the user-space applications and the Linux kernel, which is seen as a whole at this stage.

The analysis helps us to define the following controller constraints for the Linux kernel:

- **C-1**: Linux kernel should handle properly requests from user space.
- **C-2**: Linux kernel should isolate controllers address space from each other.

At this stage, the Linux kernel is seen as a single component. The next step will be to look inside it to analyze each part

with a finer grain. A first look at the internals of the Linux kernel is reported in Figure 5. This structure has been obtained following the mapping proposed by Shulyupin [27].

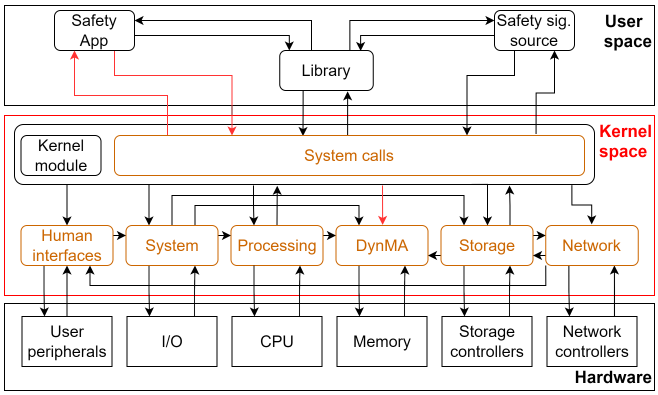

Fig. 5. Linux control structure obtained by following the mapping in [27]. Three separation levels are identified: hardware, kernel space, and user space. Arrows in red identify the interactions that are analyzed in the following sections.

In the ELISA telltale example, dynamic memory allocations are requested in indirect ways. For instance, the reset of the Watchdog is performed by writing some values into a file. This operation, as well as the `open` system call, will trigger an allocation in kernel space. Next sections will focus on the analysis of Linux kernel DynMA. In this paper we decided to apply STPA-PES to this component since ISO 26262 does not provide guidelines for its integration. In particular, it cannot be addressed with part 6 of the standard because that part copes with the development of a *new* software product. Furthermore, ISO 26262-8.12 covers the qualification of simple software components, which again is not the case of Linux kernel due to its large code-base. Finally, the proven-in-use argument of ISO 26262 cannot be made in our case because it requires that the code under analysis and its configurations are never changed. This does not hold for the Linux kernel, which is continuously evolving and provides a large number of configurations that make it difficult to find configurations similar to the system under analysis. For these reasons, we apply STPA-PES to Linux kernel DynMA to enable safety analysis for complex pre-existing software architectural elements.

B. Step1 - Linux dynamic memory allocation background

In this work, we applied the steps presented in Section III-D to Linux dynamic memory allocation as an example. The same approach can be applied to other parts of the Linux kernel, such as, for instance, the Processing sub-system (see Fig. 5).

DynMA is a Linux feature that is heavily used by many different processes and devices. In general, for a general-purpose OS it is very difficult to avoid the usage of dynamic memory. However, as reported in software safety standards [1] and software guidelines [28], it the usage of dynamic memory is not recommended in safety-critical applications. This is usually true unless the related failure modes have been exhaustively identified and the corresponding safety mechanisms are in place for fault mitigation. The safety analysis of this feature is therefore mandatory in order to understand potential criticisms that may harm the entire OS when used in safety-related applications.

In our work, knowledge about the internals of Linux's DynMA has been obtained by analyzing both the official Linux documentation [29] and other academic documentation [30]. In Linux, processes may run in two different contexts, which are named user space and kernel space. Different capabilities assigned in such spaces result in different handling strategies for dynamically allocated memory.

In user space, DynMA is responsible for managing the process heap, which is typically handled by functions within a library. Taking into consideration the `glibc` library, which is one of the most widely used in current applications, such functions are for example `malloc` , `calloc`, and `realloc` for dynamic memory allocation, while `free` is used for dynamic memory de-allocation. When the heap space is not sufficient to satisfy an allocation request, such functions may use a function called `sbrk` in order to adjust the heap size. `sbrk` is a system call provided by Linux. Other system calls that might be called by the user-space DynMA are `mmap` and `munmap` to add/create or remove, respectively, a new mapping in the virtual space of the process.

In kernel space, instead, DynMA makes use of virtual and physical memory allocators. On the lower level, physical memory is handled through an algorithm called *binary buddy allocator* [31], which manages memory blocks of power of two pages each. SLAB, SLUB, and SLOB allocators [32], [33] are instead three different algorithms implemented above the buddy allocator in order to allocate an arbitrary amount of memory. The selection of the allocator is done when Linux is built. SLAB and SLUB manage slabs, which are objects grouped in data structure called kernel caches. As a high-level level interface, the functions that are generally called to dynamically allocate memory in the kernel space are `vmalloc` and `kmalloc`. Both allocate virtually contiguous pages, but `vmalloc` allocates physically non-contiguous pages (by modifying the page tables). The corresponding functions for dynamic memory de-allocation are `vfree` and `kfree`.

C. STPA-PES applied to Linux's DynMA

The application of STPA-PES to Linux's DynMA starts from the identification of system-level losses and hazards, while the next phases involve the analysis of the decomposed SR-element.

1) Step2 - Losses and hazards: In this analysis, the Linux Kernel interactions with user space and hardware have been considered as the *system boundaries*. Interactions with the user space considered in this work are the system call while read, write, and interrupts are instead interactions between Linux and the hardware.

Losses in this stage have been derived from constraints C-1 and C-2. Therefore, the identified losses and hazards are general and suitable for the whole Kernel:

- **L-1**: Loss or damage to elements outside the system (SW elements).
- **L-2**: Loss of mission (the kernel does not correctly provide it services).

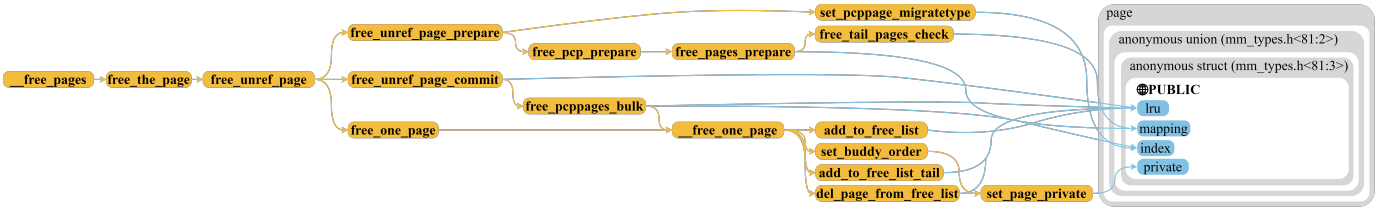

Fig. 6. Part of the graph produced by leveraging the code visualization tool for the function `vfree`. In particular, this section is related to the buddy page allocator.

The identified hazards are instead:

- **H-1**: The kernel does not respond to external input.
- **H-2**: The kernel does not maintain the integrity of the user space.
- **H-3**: The kernel does not maintain the integrity of kernel data.

Every hazard may lead to either L-1 or L-2 under different condition.

2) Step3 - DynMA element extraction: static analysis: The next phase of the STPA-PES is the SR element extraction. This work applies such a phase of the proposed approach to Linux DynMA as an example for driving the analysis to the rest of the system. Other SR features like mutual exclusion and scheduling are in the scope of sub-STPA analysis for other software elements: for this reason, such features are considered safe and correctly used in the following. In other words, once the system is defined, the context outside the boundary of the SR feature (but within the system) is assumed to be safe and correctly used. The correct use of each feature should be checked against the output of the related sub-STPA-PES. In particular, each sub-analysis provides wrong utilization patterns for each feature as output. Further details will be shown on Section IV-C5. Here, the information reported in Section IV-B is crucial for carefully selecting the functions that provide the core DynMA functionality.

This phase starts with the generation of a call graph produced by the functions mentioned in Section IV-B. A *code visualization* [34] tool can be used to obtain a graph of the dependencies among functions (and subroutines) as well as the accessed to the global context. These tools are generally capable of representing the code under analysis as a diagram (or a map) that evidences the internal dependencies and can then be used as a support for code comprehension. In this work, Sourcetrail [35] has been used for this purpose since it is free and open source. The tool has been used to analyze DynMA of Linux-5.17. In order to manage the graph complexity, the files analyzed by the tools have been carefully selected based on the knowledge of the system reported in Section IV-B. Once indexed all the symbols, the tool can be used to extract dependencies information starting from a certain function. Such an information has been obtained by using the *custom trail* feature of Sourcetrail and using nodes to represent *global variables*, *fields*, and *functions* and using edges to denote *use* and *call* relationships among them. The graph produced by the tool is a *dependency graph* that represents functions, global variables and data structure as nodes, while function call and data access are represented as edges. Such a graph contains the information of a call graph and it shows also accesses from functions to global variables and data structures. In order to reduce excessive complexity, some nodes should be hidden in the graph, starting from variables out of scope. In this work, the variables to be hidden are essentially spinlocks, which are SR variables to be considered in the scope of another SR feature analysis (e.g., mutual exclusion services). Other nodes to be hidden in the graph are edge functions that do not modify any global variable. This pruning activity allows obtaining a simplified and more readable graph that can be leveraged to abstract the control structure. Even after pruning, the whole graph obtained in this work was still too big to be displayed in this paper. Figure 6 is an excerpt of the graph focused on its part related to the chain of functions responsible to de-allocate buddy pages. Such a figure was obtained starting from the function `vfree()`. Similar dependencies were found for functions interacting with slab objects, with caches data structure, and with page tables. A dual graph has been obtained from `vmalloc()` function call, where the function acts on global data structures to allocate extra memory.

The analysis identified four groups of data structures, thus generating four controllers as the basis for the DynMA control structure. The graphs obtained by means of the code visualization tool are then used to build controller abstractions.

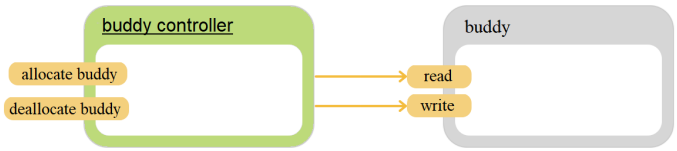

Fig. 7. Buddy controller with its interface and relative data structure to handle.

An example is reported in Figure 7, where the depicted controller is responsible for handling the buddy data structure. The figure contains other details like interfaces and relative data structure usage, while the complex call graph is hidden inside the controller. ”Allocate buddy” and ”deallocate buddy” in the interface represent respectively the access point for `alloc_pages()` and `free_pages()` function calls, and they can be used by other controllers through a control action. The buddy data structure instead offers two functions as the interface, which are read and write. The same procedure can be followed to abstract the other three controllers.

3) Step4 - Control structure: By leveraging the call graph obtained in Section IV-C2 and information obtained in Section

IV-B, this phase of STPA-PES focuses on building the control structure that abstracts each SR feature provided by the system.

By leveraging the information obtained from enforced call graphs, it is possible to draw relations among different controllers inside the control structure as shown in Figure 8, which represents the Linux DynMA abstraction. The model is coherent with the kernel structure explained in Section IV-B, where the whole address space is split between kernel and user. In the user space, "Safety element" and "Non-safety app" represent processes or code that may interact with the system under analysis. The first one has safety requirements allocated on it, while the second one is not involved in safety related functionalities. In the kernel space instead, software elements with different integrity might coexist in the same context. Since kernel space has not yet been partitioned currently into SR and NSR parts according to the state of the art, in our example we have considered a generic "kernel DynMA client" as the starting point of the DynMA request.

The control structure built until this moment represents the DynMA infrastructure that reports possible interaction among controllers. However, the model does not specify the internal behavior of each controller and possible parameters exchanged in control actions. In other words, each controller present in the model is useless. As written in step4 of Section III-D, missing information should be filled exploiting once again the system knowledge obtained in the Section IV-B. The internal behavior (process model) of *SLAB / SLUB / SLOB controller* (SLAB controller) has been extrapolated based on the above-mentioned related work [32], [33]. Specifically, such an allocator handles slab objects in carefully organized lists. When the Cache controller requests a slab to the SLAB controller, it searches within its data structure to find a suitable slab. If a free slab is available in the list, it is allocated and provided to the Cache controller. If no free slabs are available, the SLAB controller requests free memory to the Buddy controller. Instead, upon receiving a request to release a slab from the Cache controller, the SLAB controller updates its data structure and, under certain conditions, returns the memory to the Buddy controller. Allocate buddy and de-allocate buddy control actions are responsible for allocating and de-allocating a buddy, respectively, where each buddy consists of a certain number of contiguous pages. For this reason, the first control action does not require any parameter, while the second one takes as input of the buddy to be de-allocated. A similar reasoning applies for the rest of the controllers involved in the model. Note that the parameters present in the code for the function `free_pages()` are the page address and the number of pages to be de-allocated, which is different and more general with respect to a buddy. Such a decision will be better discussed in Section V-A. Next sections will move to the next step of the STPA-PES by focusing on the SLAB controller and on the *buddy controller* introduced here.

With this phase, we understand how each abstracted DynMA component interacts with each other. However, static analysis cannot catch calls performed through function pointers, which might hide important information in the call graph. To overcome this problem, we traced kernel functions called during the execution of the telltale use-case with the help of Linux Ftrace [36]. The trace revealed that no functions other than those included in the static call graph are actually called. This allowed for validating the obtained control structure.

4) Step5 - UCAs: Once built the DynMA control structure, the next phase of STPA-PES can be applied. The systematic approach starts by considering how each control action present in the model may lead to an hazard. Considering as an example control actions from the SLAB controller to the buddy controller, this section shows how to identify UCAs involved in the control structure.

The first way that classifies a control action as unsafe is in the case in which it is not provided when needed. The missing request to allocate a buddy may lead the process in execution with a wild pointer, which is a pointer that refers to memory at an arbitrary location. The usage of this pointer may write wrong values in system data structures, thus breaking the

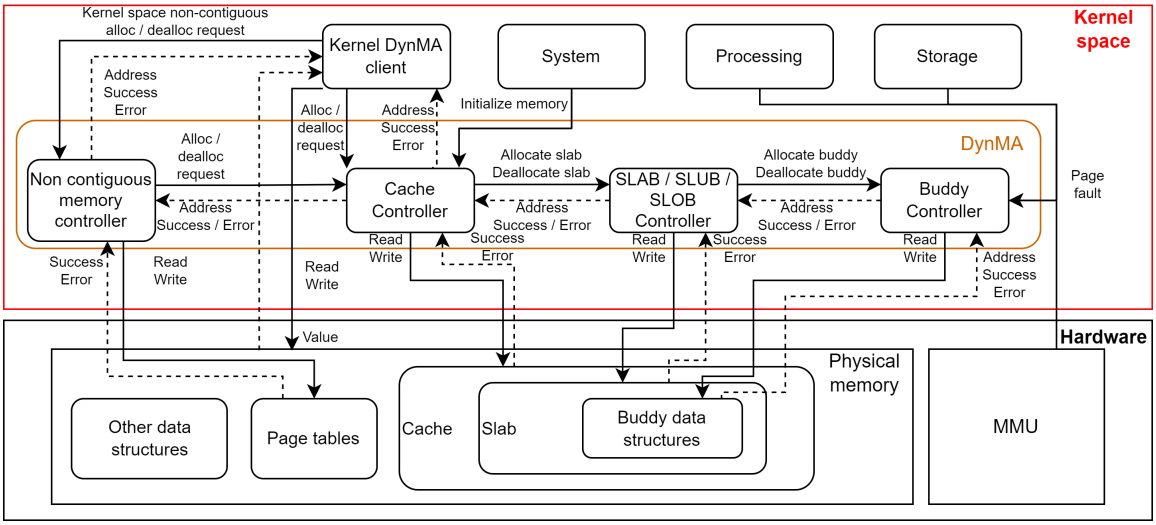

Fig. 8. DynMA control structure obtained by following the proposed approach. The DynMA controller has been further analyzed and decomposed in four controllers.

TABLE II
ALLOCATE AND DEALLOCATE BUDDY UCAS

Control action	Not provided	Provided	Too early, too late or wrong order
Allocate buddy	Not Applicable	Not Applicable	UCA-1: The Slab controller writes on the memory to be allocated before providing the alloc buddy control action [H-1, H-2, H-3] UCA-2: The Slab controller de-allocates a buddy before providing the alloc buddy control action [H-1, H-2, H-3]
Deallocate buddy	UCA-3: The Slab controller does not provide the dealloc request control action when the memory needs to be freed [H-1]	UCA-4: The Slab controller provides the dealloc buddy control action with wrong parameter (memory space outside of the memory boundaries) [H-2, H-3] UCA-5: The Slab controller provides the dealloc buddy control action twice on the same address [H-1, H-2, H-3] UCA-6: The Slab controller provides the dealloc buddy control action with a wrong parameter (buddy allocated for another process) [H-3]	Not Applicable

global context and causing an unknown behavior. However, missing allocation request alone is not a source of criticality, and for this reason, it should be considered in the ”wrong order” column. Not providing the de-allocate buddy control action once the memory is no more needed results in memory leakage, a memory space that will not be used anymore. In a system with small memory, the frequent occurrence of such an event may exhaust this resource quickly, thus blocking the entire OS.

The second case in which a control action may be unsafe is when it is provided with a wrong parameter or it is requested too many times. For the allocate buddy control action there is no parameter to check, and multiple allocations do not warn the system. For this reason, there is no way for such a control action to become unsafe in this context. Instead, in the deallocate buddy control action the buddy to be deallocated may be either inside or outside the system boundaries. In the first case, the parameter may be a buddy allocated to the process that is requesting its deallocation or to another one that still needs data contained inside. Another harmful condition is produced by the consecutive execution of the deallocate buddy control action on the same buddy, which may break the system state and cause undefined behavior.

Finally, UCAs may be categorized as provided too early, too late, or in the wrong order. For simplicity, in the example under analysis the order that has been considered as order to follow is the typical one: request an allocation, write to and read from the allocated memory, and finally de-allocate the memory. A write or a de-allocation control action provided without a previous allocation results in unpredictable behavior since such actions will use a wild pointer that may break the global context. A read operation is instead considered safe in this work since it does not modify the global state of the system. Table II reports UCAs described in this section linked together with possible hazards that each UCA may lead to.

This part of the analysis highlights the simplification of considering abstracted parameters in control actions instead of function parameters, which could have ended up with an unmanageable UCAs list. With the proposed approach, the UCAs table is manageable and understandable for the next steps.

5) Step6 - Loss scenarios: The last phase of the proposed approach should identify scenarios that lead to UCAs. For each UCA in Table II at least one scenario will be described explaining how it would lead to the specified hazard.

This phase outputs, where possible, the lines of code sequences that are responsible to generate such scenarios. Sometimes it cannot be possible for UCAs of the category ”not provided” since there are situations in which a single missed control action will not lead the system to an hazard. However, this cannot be generalized and it should be evaluated case by case. In the case of the example under analysis `alloc_pages()` and `free_pages()` function calls and read or write actions should be considered to produce such loss scenarios.

Table III shows loss scenarios produced by applying the proposed approach to UCAs presented in the previous subsection. The third column ”code example” avoids ambiguities produced by defining loss scenarios in the natural language. In general this column reports some patterns to be avoided in the kernel, like an incorrect interface usage from a driver. In this example instead the focus is in the control action from the SLAB controller to the Buddy controller, thus reporting scenarios internal to the implementation of the DynMA. It is possible to observe that for UCA-3 the third column is empty, which is due to the fact that a single missed de-allocation is not able to exhaust the memory thus blocking the system. In Scenarios 1 and 2, the two lines of codes are executed in the wrong order, thus causing access or deallocation of a memory not previously allocated. Scenario 5 modeled a double subsequent deallocation request, while the rest of the loss scenarios are related to deallocation called with wrong parameters.

Finally, this paper analyzes the consequences of control actions not executed at all. In this work, the missing execution of an allocation request has been reported as a scenario that leads to H-1. That is because a wrong or missing data structures update, or wrong address returned to the caller are the result of bugs or corrupted data structure, conditions that are out of the scope of the STPA-PES. Under these assumptions, the

TABLE III
ALLOCATE AND DEALLOCATE BUDDY LOSS SCENARIOS

UCA	Loss scenario	Code example	Hazard
UCA-1	Scenario 1: The Slab controller assumes that the buddy is already allocated, and it writes instead on a buddy that is not allocated yet [UCA-1]. As a result, the kernel uses a wild pointer that may break the system state and causing an unpredictable behavior.	*addr = value; addr = alloc_pages(order, flag);	[H-1, H-2, H-3]
UCA-2	Scenario 2: The Slab controller assumes that the buddy to be freed is already allocated while it is not, and it requests the deallocation of an unallocated memory space [UCA-2]. As a result, the kernel uses a wild pointer that may break the system state and causing an unpredictable behavior.	free_pages(addr, order); addr = alloc_pages(order, flag);	[H-1, H-2, H-3]
UCA-3	Scenario 3: The Slab controller receives a deallocate slab function from the Cache controller, but it skips the function to be performed [UCA-3]. As a result, the kernel does not respond to external input.	Not Applicable	[H-1]
UCA-4	Scenario 4: The Slab memory controller uses a memory deallocation function with an address outside the boundaries, causing a deallocation control action provided with wrong parameter (memory outside the boundary). As a result, the kernel violates the system boundaries.	high_addr = high_value; free_pages(high_addr, order);	[H-2, H-3]
UCA-5	Scenario 5: The Slab controller uses a memory deallocation function twice on the same address, causing a dealloc buddy control action provided provided twice [UCA-5]. As a result, the kernel react in an unpredictable way.	free_pages(addr, order); free_pages(addr, order);	[H-1, H-2, H-3]

UCA-6	Scenario 6: The Slab controller client uses a memory deallocation function with an address that points to a buddy allocated for another process, thus causing a deallocation control action provided with wrong parameter (buddy allocated for another process). As a result, the kernel will violate the integrity of kernel data.	Process 1
		addr = alloc_pages(order, flag);
		Process2
		free_pages(addr, order);
			[H-3]

harmful condition happens only when the allocation request does not return, thus blocking the caller from waiting for the kernel to allocate a buddy. Similar reasoning brings the missing execution of a deallocation request to be marked as a scenario that leads to H-1.

V. DERIVED REQUIREMENTS AND DISCUSSION

Based on the proposed approach, the generation of safety requirements can be guided by analyzing the loss scenarios. Code examples highlight possible design criticality or errors that shall be detected and mitigated in order to fulfill the allocated safety requirements.

Since STPA-PES can be applied to pre-exixting software, it is possible that some identified loss scenarios are covered by mechanisms already present in the software or by some specific configurations. Moreover, static checks may reduce some of the remaining harmful conditions eliminating potential systematic errors. However, it would be not enough or the software under analysis may be so complex that loss scenarios cannot be checked statically. In this case, additional safety requirements can be defined to cover the gap of remaining loss scenarios with new safety mechanisms (either inside or outside the pre-existing software).

Table IV shows requirements derived from the analysis reported in previous sections. The analysis has been completed but, as mentioned above, some of these requirements may be already satisfied by features already implemented in the pre-existing software. To support the identification of safety requirements already satisfied, the code instructions mapped in the loss scenario tables could guide the analyst to identify the ones which still need to be covered. Code instructions collected in the loss scenarios table should guide the analyst in the generation of tests useful to identify such mechanisms. In this way, new mitigation strategies will be implemented to satisfy only uncovered requirements.

A. Discussion

An important consideration to be done is related to the software element abstraction phase. In particular, the last phases of the analysis may produce different results depending on parameters defined in control actions. By following the proposed approach, such parameters are carefully selected thanks to the system knowledge. An alternative is to let control actions inherit input parameters in functions that compose each control action. This solution, however, may cause a mismatch with the control structure. An example can be represented by `free_pages()`, a function that takes in input two parameters which are the page pointer returned from `alloc_pages()` and the number of pages allocated. Such a function is involved in the deallocate buddy control action, which takes as input the buddy to be deallocated. Figure 9 evidences that the `deallocate_buddy` control action manages buddies while the `free_pages` C function manages general pages.

In this context, a loss scenario identified by a wrong parameter takes two very different meanings. In the first case, wrong parameter conditions may result in several scenarios that cannot happen inside the DynMA, for instance, the slab controller asking for the deallocation of an address that is not a buddy. Instead, the wrong abstracted parameter moves the analysis to meaningful and manageable loss scenarios.

B. Comparison with other techniques and limitations

Classic safety analysis techniques such as FTA or FMEA cannot be used for complex software. While such techniques look at the system with the finest possible grain, STPA

TABLE IV
ALLOCATE AND DEALLOCATE BUDDY DERIVED REQUIREMENTS

Loss scenario	Derived requirement	Hazard
Scenario-1	[REQ-1] The Kernel shall avoid dynamic memory usage before the correct allocation.	[H-1, H-2, H-3]
Scenario-2	[REQ-2] The Kernel shall avoid unintended deallocation requests by clients without rights.	[H-1, H-2, H-3]
Scenario-3	[REQ-3] The Kernel shall detect if Memory Allocator/Deallocator does not handle an allocation/deallocation request. In case of faults, the Kernel shall notify the requester with fault flag E_XXXX.	[H-1]
Scenario-4	[REQ-4] The Kernel shall check if all the inputs of the allocation/deallocation requests are inside the acceptable ranges. In case of inputs exceeding the allowed ranges, the Kernel shall notify the requester with fault flag E_XXXX.	[H-2, H-3]
Scenario-5	[REQ-5] The Kernel shall detect multiple consecutive deallocation requests in the wrong order. In case of faults, the Kernel shall notify the fault flag E_XXXX and shall handle only the first request.	[H-1, H-2, H-3]
Scenario-6	[REQ-4] The Kernel shall check if all the inputs of the allocation/deallocation requests are inside the acceptable ranges. In case of inputs exceeding the allowed ranges, the Kernel shall notify the requester with fault flag E_XXXX.	[H-3]

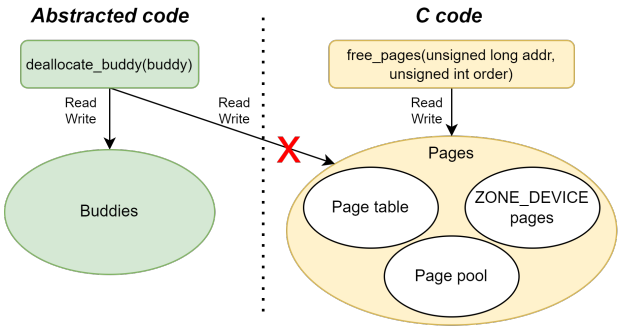

Fig. 9. Abstracted `deallocate_buddy` control action compared with `free_pages` C function. The abstracted control action shall deal with abstracted C data structure.

analyzes the system as a whole (hence with too coarse granularity). Starting from the work products obtained by applying STPA at the system level, STPA-PES looks inside the software components to understand its internals without analyzing every function in the code. In this way, STPA-PES provides the basis to bridge the gap between classic safety analysis techniques and STPA.

STPA-PES has however some limitations. While the call graph helps the analyzer in the SR element extraction phase, the build controller phase is still hard to automate. Some basic knowledge about the system is required to correctly define the controller's behavior. Moreover, the procedure to derive safety requirements from loss scenarios is case-dependent and, for this reason, also hard to generalize.

VI. CONCLUSIONS AND NEXT STEPS

This paper presented STPA-PES, an STPA-based technique suitable to drive software hazardous behavior analysis and safety requirement generation for complex pre-existing software. Concerning other STPA-based frameworks, such an approach considers the whole system in the earliest stage, while, in later phases, the decomposed SR elements are analyzed to carefully manage complexities derived from the large codebase of the software. The systematic procedure to abstract the control structure of each SR element is able to describe its model in a way that reduces the freedom and possible gaps derived from the analyzer's knowledge about the system.

The application of STPA-PES to Linux DynMA generated safety requirements that mitigation strategies should cover. The use of a code visualization tool helped in the construction of a control structure that represents the software model without the availability of the design.

Abstracting the software element control structure helps the analyst in the extrapolation of a system architecture while performing the software hazardous behavior analysis. This is particularly helpful when dealing with software implemented without following a system development lifecycle like the V-model, which guides the programmer in the definition of a clear design before starting to implement it. Moreover, the abstracted control structure gives an idea to the system analyst of the software complexity, such as a large model, a huge number of controller dependencies, or large use of global variables.

STPA-PES requires however some knowledge about the system. In particular, the controller's behavior is not modeled in the control structure, but it is hidden internally. For this reason, future refinements of the proposed technique should cope with this aspect. A way to overcome such a shortcoming is to integrate STPA-PES with finite state machine (FSM). The combination of FSM with STPA has been proposed by Xing et al. [37], but, in the context of pre-existing software, such an approach should be enforced with a code analysis technique to extrapolate the software FSM. The result will produce an approach that is able to fully represent the system without the need for excessive knowledge, e.g., automatically through the help of a tool.

ACKNOWLEDGMENT

The authors would like to thank ELISA community [2].

REFERENCES

[1] "Iso 26262:2011, road vehicles – functional safety – part 1 to 10." [Online] Available: https://www.iso.org/standard/68383.html.
[2] "Enabling linux in safety applications (elisa)." [Online] Available: https://elisa.tech/.

[3] A. Israeli and D. G. Feitelson, "The linux kernel as a case study in software evolution," *Journal of Systems and Software*, vol. 83, no. 3, pp. 485–501, 2010.

[4] S. Ravikumar and C. Subramaniam, "A survey on different software safety hazard analysis and techniques in safety critical systems," *Middle East J Sci Res*, 2016.

[5] N. G. Leveson and J. P. Thomas, "Stpa handbook," *Cambridge, MA, USA*, 2018.

[6] W. E. Vesely, F. F. Goldberg, N. H. Roberts, and D. F. Haasl, "Fault tree handbook," tech. rep., Nuclear Regulatory Commission Washington DC, 1981.

[7] M. Towhidnejad, D. R. Wallace, and A. M. Gallo, "Fault tree analysis for software design," in *27th Annual NASA Goddard/IEEE Software Engineering Workshop, 2002. Proceedings.*, pp. 24–29, IEEE, 2002.

[8] D. J. Reifer, "Software failure modes and effects analysis," *IEEE Transactions on Reliability*, vol. R-28, no. 3, pp. 247–249, 1979.

[9] M. Takahashi, Y. Anang, and Y. Watanabe, "A safety analysis method for control software in coordination with fmea and fta," *Information*, vol. 12, no. 2, p. 79, 2021.

[10] N. Leveson and J. Stolzy, "Safety analysis using petri nets," *IEEE Transactions on Software Engineering*, vol. SE-13, no. 3, pp. 386–397, 1987.

[11] A. Cherfi, M. Leeman, F. Meurville, and A. Rauzy, "Modeling automotive safety mechanisms: A markovian approach," *Reliability Engineering & System Safety*, vol. 130, pp. 42–49, 2014.

[12] I. A. Papazoglou, "Mathematical foundations of event trees," *Reliability Engineering & System Safety*, vol. 61, no. 3, pp. 169–183, 1998.

[13] M. Čepin, *Reliability Block Diagram*, pp. 119–123. London: Springer London, 2011.

[14] H. G. Lawley, "Operability studies and hazard analysis," *Chemical Engineering Progress*, vol. 70, pp. 45–56, 1974.

[15] N. Leveson, "A new accident model for engineering safer systems," *Safety Science*, vol. 42, no. 4, pp. 237–270, 2004.

[16] I. Friedberg, K. McLaughlin, P. Smith, D. Laverty, and S. Sezer, "Stpa-safesec: Safety and security analysis for cyber-physical systems," *Journal of Information Security and Applications*, vol. 34, pp. 183–196, 2017.

[17] J. Yu, S. Wagner, and F. Luo, "Data-flow-based adaption of the system-theoretic process analysis for security (stpa-sec)," *PeerJ Computer Science*, vol. 7, p. e362, Feb. 2021.

[18] A. A. A. Abdulkhaleq, *A system-theoretic safety engineering approach for software-intensive systems*. Cuvillier Verlag, 2017.

[19] N. G. Leveson, *Engineering a safer world: Systems thinking applied to safety*. The MIT Press, 2016.

[20] A. Abdulkhaleq, D. Lammering, S. Wagner, J. Röder, N. Balbierer, L. Ramsauer, T. Raste, and H. Boehmert, "A systematic approach based on stpa for developing a dependable architecture for fully automated driving vehicles," *Procedia Engineering*, vol. 179, pp. 41–51, 2017.

[21] M. Rejzek, S. H. Björnsdóttir, and S. S. Krauss, "Modelling multiple levels of abstraction in hierarchical control structures," *International Journal of Safety Science*, vol. 2, no. 01, pp. 94–103, 2018.

[22] D. Zhong, N. Wu, Q. Wang, and R. Sun, "A multi-view extended software control structure modeling and safety analysis method," in *2015 Prognostics and System Health Management Conference (PHM)*, pp. 1–5, IEEE, 2015.

[23] A. Abdulkhaleq, S. Wagner, and N. Leveson, "A comprehensive safety engineering approach for software-intensive systems based on stpa," *Procedia Engineering*, vol. 128, pp. 2–11, 2015.

[24] C. A. Thieme, A. Mosleh, I. B. Utne, and J. Hegde, "Incorporating software failure in risk analysis – part 1: Software functional failure mode classification," *Reliability Engineering & System Safety*, vol. 197, p. 106803, 2020.

[25] S. Tyszberowicz, R. Heinrich, B. Liu, and Z. Liu, "Identifying microservices using functional decomposition," in *International Symposium on Dependable Software Engineering: Theories, Tools, and Applications*, pp. 50–65, Springer, 2018.

[26] "Elisa telltale use-case." [Online] Available: https://github.com/elisa-tech/wg-automotive/blob/master/Initialy_discussed_system_scope/telltale.md.

[27] "Linux kernel map." [Online] Available: https://makelinux.github.io/kernel/map/. Accessed: 2023-11-15.

[28] M. I. S. R. Association *et al.*, *MISRA C 2012: Guidelines for the Use of the C Language in Critical Systems: March 2013*. Motor Industry Research Association, 2013.

[29] "The linux kernel." [Online] Available: https://docs.kernel.org/core-api/memory-allocation.html. Accessed: 2022-02-08.

[30] D. P. Bovet and M. Cesati, *Understanding the Linux Kernel: from I/O ports to process management*. " O'Reilly Media, Inc.", 2005.

[31] "Physical page allocation." [Online] Available: https://www.kernel.org/doc/gorman/html/understand/understand009.html.

[32] J. Bonwick *et al.*, "The slab allocator: An object-caching kernel memory allocator.," in *USENIX summer*, vol. 16, Boston, MA, USA, 1994.

[33] J. Corbet, "The slub allocator." [Online] Available: http://lwn.net/Articles/229984, 2007.

[34] S. Bassil and R. Keller, "Software visualization tools: survey and analysis," in *Proceedings 9th International Workshop on Program Comprehension. IWPC 2001*, pp. 7–17, 2001.

[35] "Sourcetrail." [Online] Available: https://www.sourcetrail.com, 2021.

[36] S. Rostedt, "Ftrace linux kernel tracing," in *Linux Conference Japan*, 2010.

[37] X. Xing, T. Zhou, J. Chen, L. Xiong, and Z. Yu, "A hazard analysis approach based on stpa and finite state machine for autonomous vehicles," in *2021 IEEE Intelligent Vehicles Symposium (IV)*, pp. 150–156, IEEE, 2021.

Raffaele Giannessi graduated at Università di Pisa and Scuola Superiore Sant'Anna in Embedded Systems. He is currently working toward the PhD degree with the Scuola Superiore Sant'Anna and he is an industrial PhD candidate of Evidence S.R.L. He focuses his research on functional safety application for complex pre-existing software.

Fabrizio Tronci graduated at Politecnico di Torino (in 2014) in Electronic Engineering - Embedded Systems. Currently he is the Functional Safety Manager of Evidence S.R.L. He is one of the italian delegate of the ISO/TC 22/SC 32/WG 8 with specific focus on ISO 26262 and PAS 8926. He conducted trainings and workshop about functional safety in italian universities and published several articles about researches on functional safety application in low level software and hardware for automotive applications.

Alessandro Biondi (IEEE Member) is Associate Professor at the Real-Time Systems (ReTiS) Laboratory of the Scuola Superiore Sant'Anna. He graduated (cum laude) in Computer Engineering at the University of Pisa, Italy, within the excellence program, and received a Ph.D. in computer engineering at the Scuola Superiore Sant'Anna under the supervision of Prof. Giorgio Buttazzo and Prof. Marco Di Natale. In 2016, he has been visiting scholar at the Max Planck Institute for Software Systems (Germany). His research interests include design and implementation of real-time operating systems and hypervisors, schedulability analysis, cyber-physical systems, synchronization protocols, and component-based design for real-time multiprocessor systems. He was recipient of six Best Paper Awards, one Outstanding Paper Award, the ACM SIGBED Early Career Award 2019, and the EDAA Dissertation Award 2017.