Towards Mapping Control Theory and Software Engineering Properties using Specification Patterns

Ricardo Caldas and Razan Ghzouli
Chalmers | U. of Gothenburg, Sweden

Alessandro V. Papadopoulos
Mälardalen University, Sweden

Patrizio Pelliccione
Chalmers | U. of Gothenburg, Sweden
Gran Sasso Science Institute, Italy

Danny Weyns
KU Leuven, Belgium
Linnaeus U., Sweden

Thorsten Berger
Chalmers | U. of Gothenburg, Sweden
Ruhr University Bochum, Germany

***Abstract*—A traditional approach to realize self-adaptation in software engineering (SE) is by means of feedback loops. The goals of the system can be specified as formal properties that are verified against models of the system. On the other hand, control theory (CT) provides a well-established foundation for designing feedback loop systems and providing guarantees for essential properties, such as stability, settling time, and steady state error. Currently, it is an open question whether and how traditional SE approaches to self-adaptation consider properties from CT. Answering this question is challenging given the principle differences in representing properties in both fields. In this paper, we take a first step to answer this question. We follow a bottom up approach where we specify a control design (in Simulink) for a case inspired by Scuderia Ferrari (F1) and provide evidence for stability and safety. The design is then transferred into code (in C) that is further optimized. Next, we define properties that enable verifying whether the control properties still hold at code level. Then, we consolidate the solution by mapping the properties in both worlds using specification patterns as common language and we verify the correctness of this mapping. The mapping offers a reusable artifact to solve similar problems. Finally, we outline opportunities for future work, particularly to refine and extend the mapping and investigate how it can improve the engineering of self-adaptive systems for both SE and CT engineers.**

***Keywords*-Self-adaptive systems, feedback loops, control theory, properties, mapping of properties.**

I. Introduction

Providing evidence that a self-adaptive system behaves according to the stakeholders' requirements is challenging [1]–[3], especially when the system operates in uncertain environments. A potential remedy is to exploit principles from control theory (CT) to engineer self-adaptive systems [4]. Control theory provides a mathematical framework for designing and analyzing dynamic systems, offering a formal basis to provide guarantees for control properties such as stability, overshoot, and settling time. For computing systems, CT has primarily been used to manage low-level resources, such as CPU cycles, communication bandwidth, and hardware [5]. Recently, there has been an increasing interest in applying the mathematical framework of CT to control software elements [6]–[10] as well.

A common approach in software engineering (SE) to provide guarantees for a self-adaptive system is to test it against its requirements, or formally, by specifying formal properties[1] that are verified against models of the self-adaptive system [12]–[14]. Such formal approaches work case by case using tools such as model checkers. On the other hand, CT-based solutions provide *guarantees by design*—that is, controller models properly designed according to the mathematical principles of CT satisfy the target CT properties. Yet, understanding how, and to what extent, traditional SE approaches to self-adaptation consider and comply with properties from CT is challenging given the principle differences between the two paradigms.

In this paper, we take a first step towards determining whether and how traditional SE approaches to self-adaptation consider properties from CT. The relationship between SE approaches to self-adaptation and CT has been investigated from different angles. One line of research—reflected in Brun et al. [12] and Filieri et al. [15], among others—determines the mapping of the elements of a CT feedback loop design to the elements of the MAPE-K architecture [16]. Another line of research—reflected in Shevtsov et al. [17] and Caldas et al. [18], among others—studies the synthesis of controllers for correct and efficient adaptation based on control theory. Recently, Cámara et al. [19] made a step forward in bridging the gap by proposing a mapping between CT properties and self-adaptive systems properties relying on a common language. We stand on their shoulders to investigate the foundational issue of whether and to what extent properties of traditional SE approaches to self-adaptation consider properties from CT.

Inspired by a concrete case—the Simulink-model-based specification of a control design by Scuderia Ferrari (F1) [20]—the engineering process used in our work includes adaptation goal identification, CT design, controller implementation and integration, and finally validating the implementation against the goals. Once the goals include CT properties, the challenge arises to check whether the implementation satisfies those CT properties. Here, the interplay between SE and CT takes place. For instance, suppose that the software engineers introduce a fault in the software when optimizing C code, and suppose that this

[1]A property expresses what a system should or should not do, or how a system should or should not behave. To analyze a property, we need a property specification [11] that for instance can be verified against a model. In this paper, the term *property* refers to formally specified properties.

© 2021 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

fault induces behavior that violates the requirement of keeping a safe distance from the vehicle ahead, previously guaranteed by the controller design. How can the test engineers verify that the safety requirements considered by the control engineers in the design still hold? To enable engineers to check CT properties on the implementation, we envision a mapping between CT properties and SE properties (here formulated in LTL). A formulation of CT properties in some kind of temporal logic might enable, for instance, the use of model checkers (e.g., DIVINE [21] or Uppaal [22]), monitoring techniques (e.g., Larva [23] or PREDIMO [24]), and model-based testing [25]. To that end, we perform initial steps towards determining the mapping by exploiting so-called property specification patterns [26], [27]. We show the overall engineering process and mapping through an example of adaptive cruise control. We also check the defined mapping between CT and SE properties by investigating whether the LTL formula (SE property) presents the same behaviors that can be observed at the CT simulation level (CT property). To that end, we use the model checker DIVINE [21]. The results of the validation are promising, indicating feasibility and effectiveness of identified mappings between CT and SE properties. We provide an online replication package [28], including our artifacts (e.g., Simulink model) and further details.

II. Background and Motivation

Engineering Process. Engineering a controller to realise self-adaptation typically involves an iterative process where control experts work separately from software engineers [29]. Such a process usually comprises six steps: (1) identify the adaptation goals, (2) identify the knobs (a.k.a., configuration options or calibration parameters), (3) devise the system model, (4) design the controller, (5) implement and integrate the controller, and (6) test and validate the system [15]. Especially step (5), the implementation and integration of controllers into a larger system, is considered a challenging task [30]. In principle, engineers follow a model-driven approach [31], [32], relying on a variety of tools and modeling languages (e.g., (MATLAB/Simulink, Modelica, SCADE), where the control model is transformed into hardware- and environment-specific models and eventually into source code. Still the generated code, as we learned from the Ferrari case, needs to be further optimized and customized before it can be integrated into a larger system. This requires extensive manual modification of the code [33]—an error-prone activity that may introduce bugs that are hard to spot without proper verification techniques. In addition, when the controller becomes part of a larger system, it is also influenced by it, further questioning whether the necessary CT properties still hold. Apparently, verifying code requires SE verification techniques, but to what extent the system does (or can) address CT properties is an open question.

CT versus SE Properties. Mapping and comparing SE and CT properties is challenging. The former are typically formulated over a model of the system (often graph-based, such as a finite or a Büchi automaton) and expressed in a formal language, such as in temporal logics or in a process algebra. The latter are completely defined in algebraic formulations. While challenging, we believe that a mapping between the properties of the two worlds might be facilitated by expressing them in a common notation. Current approaches, such as Cámara et al. [19], use a formal language as common notation. Although flexible, specifying properties using a common formal language can be time-consuming and prone to errors.

Process and Properties at Ferrari. We further illustrate this problem based on the process followed by Scuderia Ferrari [20]. The manufacturer develops software for high-performance racing cars under tight one-week sprints. A typical requirement to be addressed from the control software involves four engineers: control experts, software developers, testers, and track engineers. At the highest level of development, control experts specify controller behavior using block diagrams aiming to guarantee the required control properties (e.g., stability, settling time, overshoot). The controller specifications are transformed into lower-level C code, where software engineers apply customizations and optimizations. The modified code is then thoroughly tested using software-in-the-loop (SIL) or hardware-in-the-loop (HIL) techniques. At the testing level, testers and control experts monitor the simulated behavior to guarantee that it behaves according to the design, i.e., that the car behavior complies with the control properties. Finally, the control software is deployed. At runtime, the track application engineers constantly monitor, elicit bug-fixing or improvement requirements, and in emergency cases perform hot-fixes to the running vehicle. The pipeline is executed in 7 days of intense work, from which half is concerned with specification, two with coding, one with testing, and the last three with validation.

Mapping Properties using Specification Patterns. As motivated above, understanding the extent to which properties considered by traditional SE approaches to self-adaptation cover fundamental properties of CT urges for a mapping between the two. To this end, we propose a technique that uses patterns to map CT properties to SE properties. Inspired by the process and details of how Ferrari engineers controllers, in this short paper, we demonstrate the feasibility and effectiveness of mapping CT and SE properties. Specifically, we design an example control model in Simulink with guarantees for CT properties. The model is then transformed into C code and is subject to modifications. Next, we define properties and verify whether the control properties still hold at code level. To consolidate the solution, we map the properties in both worlds using specification patterns as a common language, and we verify the correctness of this mapping. In the next sections III to V we explain this process using a running example.

III. System Model Design

The process starts with the design of a control model that complies to specific requirements. Using Simulink, we designed an adaptive cruise control (ACC), a driver assistance system, for a vehicle (ego vehicle). We modeled a scenario where the ego vehicle is provided with an ACC that automatically tracks a set velocity and adjusts the ego vehicle speed to maintain a safe distance from a preceding vehicle (lead vehicle) [34]. The

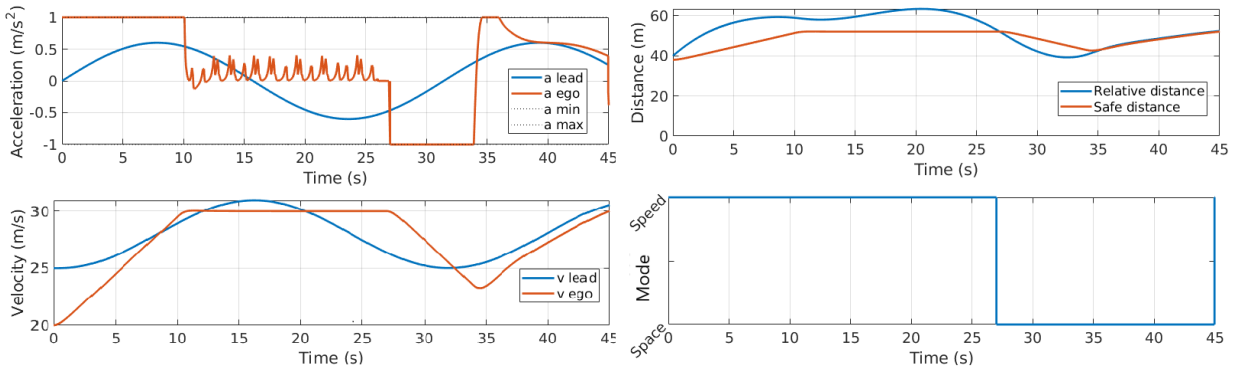

Fig. 1. Simulation excerpt of the designed CT model. Left: relation between acceleration, and leading and ego vehicles' speed. Right: relation between D_{rel} and D_{safe}, and the switching behavior between space mode and speed mode controllers.

ACC system has two operating modes: (1) Speed control where the ego vehicle follows a given speed (V_{set}) and (2) spacing control to keep a safe distance from the lead vehicle (D_{safe}).

We used a model-based approach to design two proportional–integral–derivative (PID) controllers, speed mode PID and space mode PID, that guarantee the requirement: "If the relative distance between the two vehicles (D_{rel}) is less than a safe distance (D_{safe}), the controller of the ego vehicle should adjust its speed (V_{ego}) for D_{rel} to become greater than D_{safe}, otherwise follow a given velocity (V_{set})." In other words, if the safety of the vehicle is breached, the controller should adjust the vehicle's speed to maintain a safe distance. The relative distance (D_{rel}) is the difference between the ego vehicle's position (x_{ego}) and the lead vehicle's position (x_{lead}), see Eq. (1). The safe distance is a function of the ego vehicle velocity, Eq. (2), where $D_{default}$ is the standstill default spacing and $T_{gap} \times V_{ego}$ is the gap between the vehicles. T_{gap} is chosen to enable the ego vehicle to break without crashing the leading vehicle [35].

$$D_{rel} = x_{lead} - x_{ego} \tag{1}$$

$$D_{safe} = D_{default} + T_{gap} \times V_{ego} \tag{2}$$

The switching between the two PIDs depends on a velocity error (e_v) and a distance error (e_d), see Eq. (3). We have used a state-based notation that produces the activation signal (1 for speed mode and -1 for space mode) followed by a switch to implement the mode switching in Simulink. With the switch the two PIDs do not operate simultaneously, satisfying the safety requirements.

$$Mode = \begin{cases} 1 \text{ speed mode} & \text{if } e_v = V_{set} - V_{ego} < 0 \\ -1 \text{ space mode} & \text{if } e_d = D_{safe} - D_{rel} > 0 \end{cases} \tag{3}$$

Figure 1 shows an excerpt of the simulation showing the behavior of the designed controllers for 45 sec. In this setting, we simulated a scenario where the speed of the lead vehicle varies in time according to a sine wave resulting in changes in the distance between the two vehicles. The ACC switches between the space controller and the speed controller to keep the ego vehicle safe. At the moment 27 sec a breach in the safety requirement occurs resulting in the ACC reacting to keep safety. The reaction must lead into a state where the system settles around an equilibrium point. If this happens, the system is said to be stable.[2]

IV. Modeling Stability as a SE Property

Guaranteeing that the system is stable at code-level (as illustrated in Fig. 1), requires that the property is modeled in a formal language such as a graph-based formulation, temporal logic, or process algebra. In this section we explain how we modeled stability using linear temporal logic.

A control system is *stable* even if the error $e(t)$ is not converging to zero, but the error is bounded. More specifically, in control terms, if the initial value of the system output is "close" to the equilibrium value, then the evolution over time of the output of the system will be bounded [19].

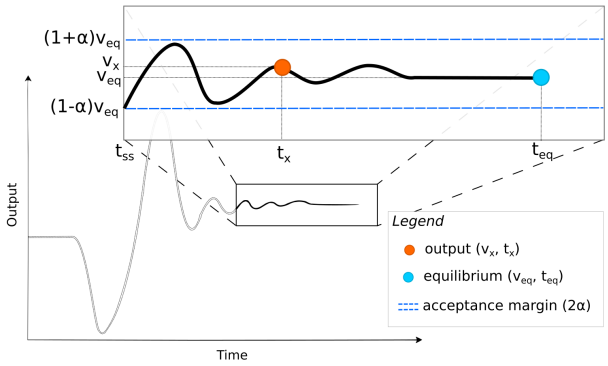

Fig. 2. Illustration of the CT property Stability for a step input.

Importantly, at some point the system must reach a so-called steady-state, where the signal is bounded. Therefore, we formalize steady-state beforehand. Let us suppose that a system is stimulated by a unitary step. Given the current state of the system's response output (v_x), the equilibrium value (v_{eq}) estimated with the output samples and the acceptance margin (α) for convergence. With this in mind we formulate the steady-state condition (SS).

$$SS \equiv |v_x - v_{eq}| \leq \alpha \tag{4}$$

[2]There are techniques for the analysis of the stability of hybrid systems based on the physical model of the system, see [36], but the description of such techniques is beyond the scope of the paper.

Equation (4) asserts that the steady-state condition holds when the distance between the current value and the equilibrium value is bounded by the acceptance margin.

The stability property is defined by the ability to reach and stay at steady-state. The output measurement of stable systems converges to steady-state if the system is 'excited' by an input. Figure 2 illustrates the response to a step input where the set of output values (v_x) is within a band defined by the acceptance margin (α). The acceptance margin is a limiting value relative to the equilibrium point (v_{eq}).

How to formulate the CT stability property in LTL? We exploit specification patterns as a common language to create the mapping between CT and SE properties. A specification pattern [26], [27] is defined as a tuple <*Scope, Pattern*>. A Scope determines the *extent of a program execution over which the pattern holds*[3]. Examples of scopes are Globally (entire execution trace) and After X (execution trace after a state/event X). A Pattern describes a *generalized recurring system attribute* [27]. Examples of patterns are Universality (a property that always holds) and Existence (a property that eventually holds). A property can be specified by one specification pattern or a composition of multiple patterns using a nesting operator. For instance, the Universality and the Existence patterns might be composed under the Globally scope to obtain the Globally, Universality Existence (i.e., $\Box\Diamond P$).

We rely on a syntactical comparison of CT and SE properties to map them using specification patterns. In our running example, we analyze the stability property from the two viewpoints. Stability as a CT property is represented by a feedback loop in which the ego's actual position is fed back into the controller. The ego's position is used to calculate the relative difference between vehicles and determine which acceleration will restore ego's safe position. In other words, stability determines the ability of the ego vehicle to restore and maintain itself in a wanted state for undetermined time.

Globally Untimed Existence Universality
Globally eventually always SS **holds.**

We formalize the Stability property by employing the specification pattern Globally Existence, which aims at describing that events/states eventually holds, nested with Globally Universality specifying the case that the events/states always hold. Syntactically, the Globally Existence Universality coincides with the stability property. The Globally Existence Universality is represented in temporal logic as shown in Eq. (5).

$$\textit{Stability} \equiv \Diamond(\Box SS) \tag{5}$$

In our running example, the ego vehicle must restore its position beyond the safety distance whenever the ego vehicle gets too close to the leading vehicle and maintains the safe distance. Therefore, we defined the steady-state as a function of Eqs. (1) and (2), where $v_x = D_{\text{rel}}$, $v_{\text{eq}} = D_{\text{safe}}$, and $\alpha = 0.05 \times D_{\text{safe}}$. Lastly, stability in the running example's context can be formulated as shown in Eq. (6).

[3]https://matthewbdwyer.github.io/psp/patterns/scopes.html

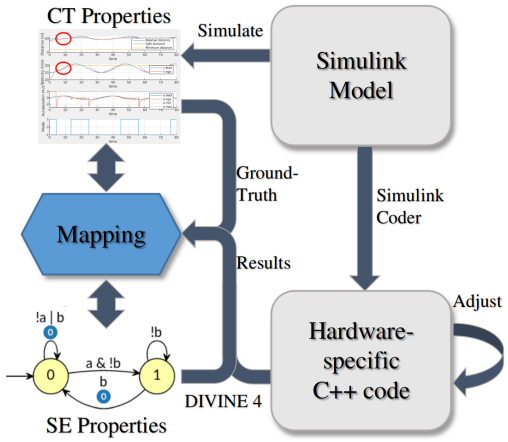

Fig. 3. Using DIVINE 4 to get confidence on the CT-SE properties mapping.

$$\textit{Stability} \equiv \Diamond(\Box(D_{\text{rel}} - D_{\text{safe}} > \alpha)) \tag{6}$$

It is important to note the absence of the absolute operation around $D_{\text{rel}} - D_{\text{safe}}$. In this specific case, when the distance between ego and the leading car is greater than $\alpha = 0.05 \times D_{\text{safe}}$ the safety requirement is not violated and the system is considered stable.

V. Checking the Mapping

In this section, we use model checking to increase the confidence about the correctness of the mapping[4]. To that end, we model check the SE property with three different scenarios in which we can attest whether the requirement is satisfied or not by visualizing the CT simulation output. Thus, we check the correctness of the mapping by comparing whether the LTL formula (SE property) holds in comparison with the expected result observed in the CT simulation level (CT property). As exemplified by Figure 3, in simulation, we capture the former three scenarios behavior using Simulink. We auto-generate the C++ code for the System model using Simulink Coder[5] for each scenario. Using DIVINE 4[6] we assure the correctness of our mapping by feeding both the generated Büchi Automata in never claim representation[7] (SE property) and the generated C++ code (CT property behavior) to the model checker.

The scenarios in Table I are tailored to show whether there is a semantic equivalence between safety at CT level and SE level. Therefore, they need to explore the different behavior that might occur during system execution. To generate the scenarios, we changed the initial settings of the experiment with different vehicle's starting positions (x_0) and starting speeds (v_0).

We ran the simulation and model checked all scenarios on a Ubuntu 18.04, processor Intel(R) Core(TM) i7-8665U CPU @

[4]Instructions on how to replicate our experiments as well as the technicalities of the model checking process are available in our online appendix [28].

[5]https://se.mathworks.com/products/simulink-coder.html

[6]https://divine.fi.muni.cz/index.html

[7]The Büchi Automata in never claim was manually encoded within DIVINE.

TABLE I
SCENARIOS

ID	v_0 ego	v_0 lead	x_0 ego	x_0 lead	Description
Case 1	10 km/h	30 km/h	10 m	50 m	Ego is always at a safe distance.
Case 2	20 km/h	25 km/h	3 m	5 m	Ego recovers from unsafe distance.
Case 3	40 km/h	15 km/h	10 m	20 m	Ego cannot recover from unsafe distance.

1.90GHz, and 32GB memory. All the scenarios returned the expected result, the Stability property (Eq. (6)) holds for cases 1 and 2 but not for case 3, see Table II.

TABLE II
MODEL CHECKING RESULTS

ID	Mem. Used	Exec. Time	Ground-Truth	Result
Case 1	323.9 MB	3.70sec	true	true
Case 2	328.5 MB	3.59sec	true	true
Case 3	326.4 MB	3.18sec	false	false

The model checking results[8] return either *Error found* or *No error found*. In our case, *Error found* translates to *true* in Table II, and the other way around for *Error found*. Such translation results from how checking liveness in DIVINE 4 and how we have implemented the Stability property.

VI. RELATED WORK

We start with related work that discusses properties of self-adaptive systems from a SE perspective. Then we discuss related work that emphasizes the importance of relating traditional SE properties to CT properties.

Back in 2012, the authors of [37] performed a systematic literature review on the use of formal methods in self-adaptive systems. The results show that safety, liveness, and reachability are the main properties considered and these properties are primarily used to verify the efficiency/performance, reliability, and functionality of self-adaptive systems. While instrumental for SE properties considered in self-adaptive systems, the authors do not look into a mapping with CT properties.

The community papers [1] and [2] that emerged from a Dagstuhl seminar state that assuring that a self-adaptive system complies with its requirements requires an enduring process that spans the whole lifetime of the system. The authors refer to this process as "perpetual assurances" and emphasize that control theory offers a basis to design solutions that provide such assurances.

The study in [38] is a pioneering work mapping between quality attributes of self-adaptive systems and properties of control theory. For instance, performance with latency and throughput is mapped to settling time. The proposed mapping is done based on terminology derived from a literature survey. In [7], the authors derived a mapping between software qualities and control properties based on the results of a systematic literature review on control-theoretical software adaptation. For instance, guaranteeing settling time may be associated with most software qualities since the property refers to guarantees on the time it takes to bring measured quality property close to its goal. In both papers, the presented mapping stayed at a high level in comparison to our work.

The most relevant paper for the work presented in this paper is [19]. In that paper, the authors mapped key properties that characterize self-adaptive systems to control properties, leveraging the formalization of both in temporal logic. While that work relies on a general formal notation to identify the mapping between SE and CT properties, our work leverages on the structures of a set of established patterns.

The paper [39] highlights recent efforts on self-stabilization in aggregate computing, for instance [40]–[42]. These efforts focus on providing guarantees for control-based properties to algorithms for self-organization of distributed systems, in contrast to mapping how properties are considered in both SE and CT. Furthermore, such effort reinforces the promising path of using of CT tooling for safety assurance provision for aggregate algorithms.

VII. CONCLUSION AND FUTURE WORK

To improve the engineering of self-adaptive systems, we proposed a technique to unify the properties used in CT design and those used in SE verification. Our technique relies on using existing specification patterns as a common notation. To that end, we follow a bottom-up approach inspired by Scuderia Ferrari to provide evidence for safety requirements from both CT and SE viewpoints. The properties are mapped using property specification patterns as a common language. Such properties are formalized in LTL and fed into DIVINE 4 to consolidate the proposed mapping through model checking. Our initial results are promising based on the mapping.

Future work should identify and map further properties and extend the exploration space of our validation. Our systematic approach for validation of our mapping provides a pathway for building evidence that the mapping is sound. Specifically, we believe the CT properties settling time, overshoot, steady-state error should be mapped to the properties widely used for verification for self-adaptation, e.g., reachability, security, privacy, availability. In this work, we considered a CT property that does not require explicit time and for this reason it was enough to map it to untimed property specification patterns, i.e. those initially proposed in [26]. It is worth mentioning that there exist also properties specification patterns with explicit time as well as with probability [27], which might be exploited for other CT properties, when needed. It would be also interesting to extend, if needed, the catalog of specification patterns with specific patterns that are tailored to CT properties. We believe that the mapping could be an important contribution to both the SE and CT communities, by giving, on one side, a concrete instrument for engineering trustworthy and safe autonomous

[8]See folder 'reports' in our online appendix [28]

system, and, on the other side, it might facilitate the cross-fertilisation among the two communities. Another direction for future research concerns the investigation of the topics of this paper with companies producing autonomous systems involving both CT controllers and software produced by developers.

ACKNOWLEDGMENT

This work is supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation. This work is also partly supported by the Swedish Research Council (VR) via the "PSI" project. The authors also acknowledge financial support from Centre of EXcellence on Connected, Geo-Localized and Cybersecure Vehicle (EX-Emerge), funded by Italian Government under CIPE resolution n. 70/2017 (Aug. 7, 2017).

REFERENCES

[1] R. De Lemos *et al.*, "Software engineering for self-adaptive systems: Research challenges in the provision of assurances," in *Software Engineering for Self-Adaptive Systems III. Assurances*. Springer, 2017.

[2] D. Weyns *et al.*, "Perpetual assurances for self-adaptive systems," in *Software Engineering for Self-Adaptive Systems III. Assurances*. Springer, 2017.

[3] D. Weyns, "Introduction to self-adaptive systems: A contemporary software engineering perspective." Wiley, 2020.

[4] Hellerstein *et al.*, *Feedback control of computing systems*. Wiley Online Library, 2004, vol. 10.

[5] Xiaoyun Zhu, Zhikui Wang, and S. Singhal, "Utility-driven workload management using nested control design," in *2006 American Control Conference*, 2006, pp. 6 pp.–.

[6] A. Filieri, H. Hoffmann, and M. Maggio, "Automated design of self-adaptive software with control-theoretical formal guarantees," in *36th International Conference on Software Engineering*. ACM, 2014.

[7] S. Shevtsov *et al.*, "Control-theoretical software adaptation: A systematic literature review," *IEEE Transactions on Software Engineering*, vol. 44, no. 8, pp. 784–810, 2018.

[8] M. Maggio *et al.*, "Automated control of multiple software goals using multiple actuators," in *11th Joint Meeting on Foundations of Software Engineering*. ACM, 2017.

[9] K. Angelopoulos *et al.*, "Engineering self-adaptive software systems: From requirements to model predictive control," *ACM Transactions on Autonomous and Adaptive Systems*, vol. 13, no. 1, 2018.

[10] M. Maggio *et al.*, "Control-system stability under consecutive deadline misses constraints," in *32nd Euromicro Conference on Real-Time Systems*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

[11] L. Lamport, "Who builds a house without drawing blueprints?" *Communications of the ACM*, vol. 58, no. 4, pp. 38–41, 2015.

[12] Y. Brun *et al.*, *Engineering Self-Adaptive Systems through Feedback Loops*. Springer, 2009.

[13] M. U. Iftikhar and D. Weyns, in *International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, 2014.

[14] R. Calinescu *et al.*, "Engineering trustworthy self-adaptive software with dynamic assurance cases," *IEEE Transactions on Software Engineering*, vol. 44, no. 11, pp. 1039–1069, 2017.

[15] A. Filieri *et al.*, "Software engineering meets control theory," in *IEEE/ACM 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. IEEE, 2015.

[16] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, no. 1, pp. 41–50, 2003.

[17] S. Shevtsov, D. Weyns, and M. Maggio, "Handling new and changing requirements with guarantees in self-adaptive systems using simca," in *IEEE/ACM 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, 2017.

[18] R. D. Caldas *et al.*, "A hybrid approach combining control theory and ai for engineering self-adaptive systems," in *IEEE/ACM 15th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, 2020.

[19] J. Cámara *et al.*, "Towards bridging the gap between control and self-adaptive system properties," in *IEEE/ACM 15th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, 2020.

[20] C. Silenzi, "Software Engineering in Ferrari F1," in *37th International Conference on Software Engineering - Volume 1*. IEEE Press, 2015.

[21] Z. Baranová *et al.*, "Model checking of C and C++ with DIVINE 4," in *Automated Technology for Verification and Analysis*, ser. LNCS, vol. 10482. Springer, 2017, pp. 201–207.

[22] J. Bengtsson *et al.*, "UPPAAL — a Tool Suite for Automatic Verification of Real–Time Systems," in *Workshop on Verification and Control of Hybrid Systems III*, ser. LNCS, no. 1066. Springer–Verlag, 1995.

[23] C. Colombo, G. J. Pace, and G. Schneider, "Larva — safer monitoring of real-time java programs (tool paper)," in *7th IEEE International Conference on Software Engineering and Formal Methods*, 2009.

[24] P. Zhang *et al.*, "Automatic generation of predictive monitors from scenario-based specifications," *Information and software technology*, vol. 98, pp. 5–31, 2018.

[25] H. G. Gurbuz and B. Tekinerdogan, "Model-based testing for software safety: a systematic mapping study," *Software Quality Journal*, vol. 26, no. 4, pp. 1327–1372, 2018.

[26] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, "Patterns in property specifications for finite-state verification," in *21st International Conference on Software engineering*, 1999, pp. 411–420.

[27] M. Autili *et al.*, "Aligning qualitative, real-time, and probabilistic property specification patterns using a structured english grammar," *IEEE Transactions on Software Engineering*, vol. 41, no. 7, pp. 620–638, 2015.

[28] R. Caldas and R. Ghzouli, "Online appendix," 2021. [Online]. Available: https://doi.org/10.5281/zenodo.5201544

[29] J. Liu *et al.*, "Actor-oriented control system design: A responsible framework perspective," *IEEE Transactions on Control Systems Technology*, vol. 12, no. 2, pp. 250–262, 2004.

[30] J. L. Hellerstein, "Engineering autonomic systems," in *Proceedings of the 6th international conference on Autonomic computing*, 2009, pp. 75–76.

[31] P. Neis, M. A. Wehrmeister, and M. F. Mendes, "Model driven software engineering of power systems applications: literature review and trends," *IEEE Access*, vol. 7, pp. 177 761–177 773, 2019.

[32] J. Schaefer *et al.*, "Future automotive embedded systems enabled by efficient model based software development," SAE Technical Paper, Tech. Rep., 2021.

[33] F. Křikava *et al.*, "Contracts-based control integration into software systems," in *Software Engineering for Self-Adaptive Systems III. Assurances*. Springer, 2017.

[34] G. Marsden, M. McDonald, and M. Brackstone, "Towards an understanding of adaptive cruise control," *Transportation Research Part C: Emerging Technologies*, 2001.

[35] T.-W. Lin, S.-L. Hwang, and P. A. Green, "Effects of time-gap settings of adaptive cruise control (acc) on driving performance and subjective acceptance in a bus driving simulator," *Safety science*, 2009.

[36] R. Goedel, R. G. Sanfelice, and A. R. Teel, "Hybrid dynamical systems: modeling stability, and robustness," 2012.

[37] D. Weyns *et al.*, "A survey of formal methods in self-adaptive systems," in *5th International C* Conference on Computer Science and Software Engineering*, 2012.

[38] N. M. Villegas *et al.*, "A framework for evaluating quality-driven self-adaptive software systems," in *6th International Symposium on Software engineering for Adaptive and Self-managing Systems*, 2011.

[39] M. Viroli *et al.*, "From distributed coordination to field calculus and aggregate computing," *Journal of Logical and Algebraic Methods in Programming*, vol. 109, no. 2019, p. 100486, 2019.

[40] S. Dasgupta and J. Beal, "A Lyapunov analysis for the robust stability of an adaptive Bellman-Ford algorithm," *2016 IEEE 55th Conference on Decision and Control, CDC 2016*, no. Cdc, pp. 7282–7287, 2016.

[41] M. Viroli *et al.*, "Engineering resilient collective adaptive systems by self-stabilisation," *ACM Transactions on Modeling and Computer Simulation*, vol. 28, no. 2, 2018.

[42] Y. Mo *et al.*, "A resilient leader election algorithm using aggregate computing blocks," *IFAC-PapersOnLine*, vol. 53, no. 2, pp. 3336–3341, 2020.