Ocean Engineering 270 (2023) 113685

Contents lists available at ScienceDirect

Ocean Engineering

journal homepage: www.elsevier.com/locate/oceaneng

Towards contract-based verification for autonomous vessels

Tobias Rye Torben [a,*], Øyvind Smogeli [a,b], Jon Arne Glomsrud [c], Ingrid B. Utne [a], Asgeir J. Sørensen [a]

[a] Centre for Autonomous Marine Operations and Systems (AMOS), Norwegian University of Science and Technology (NTNU), Trondheim, Norway
[b] Zeabuz AS, Trondheim, Norway
[c] Group Research and Development, Det Norske Veritas (DNV), Høvik, Norway

ARTICLE INFO

Keywords:
Autonomous vessels
Contract-based design
Verification
Formal methods
Simulation-based testing

ABSTRACT

Design and verification of autonomous vessels represent a major interdisciplinary engineering challenge due to the combination of high system complexity and the interaction with dynamic, uncertain, and unstructured environments. This paper investigates the use of contract-based methods to address both design and verification challenges of control systems for autonomous vessels. The paper first presents a formal framework for specification of components and assume-guarantee contracts using the syntax of the Z3 automated theorem prover. Then, the paper proposes a methodology for contract-based verification using the formal framework. The methodology is divided into 4 steps: (1) Hazard identification between the autonomous vessel and the operative environment in order to define the top-level component and contract, (2) stepwise refinement of the top-level component into detailed sub-components and contracts, (3) definition of test setups for simulation-based testing to verify that components meet their contract, and (4) applying a recursive procedure for contracts-based system verification. The framework and methodology are demonstrated the in a case study with an autonomous passenger ferry.

1. Introduction

The control systems that govern modern ships are continuously increasing in complexity and becoming more software intensive. The recent development of autonomous vessels represents the ultimate culmination of this trend. Building on top of existing advanced maritime guidance, navigation, and control systems, autonomous vessels must additionally be capable of obtaining situational awareness and performing intelligent planning and decision making under uncertainty in the dynamic and unstructured maritime environment. The development of autonomous vessels is enabled by recent advances in control, optimization, planning, artificial intelligence, computer vision, and sensor fusion, in addition to the ever-increasing computational resources available for embedded systems. However, although these technological advances allow us to implement the different functionalities necessary for autonomy, combining all of them into an integrated system, layered on top of the already complex maritime control systems, results in an unprecedented level of system complexity. In addition to the sheer complexity, verification of autonomous systems is generally very hard due to their extensive sensing and interaction with the open environment. This leads to an infinite number of unknown scenarios that an autonomous vessel may encounter in operation, and it will thus never be possible to specify the required behavior in every scenario during design (Torben et al., 2022b; Murray et al., 2022). To design reliable and robust autonomous vessel control systems and verify their safety and functionality, there is a clear need for new methodology both in design and verification.

Several recent works propose methodologies for verification of autonomous vessel control systems. The use of Systems-Theoretic Process Analysis (STPA) has been proposed for deriving safety requirements to verify against (Rokseth et al., 2019; Chaal et al., 2020). The use of STPA in combination with Bayesian Belief Networks has also been proposed as a design methodology to give autonomous vessels an online risk management capability (Utne et al., 2020). Simulation-based testing is frequently proposed as a methodology to produce verification evidence for autonomous vessels (Pedersen et al., 2020). Existing work addresses simulation-based testing of collision avoidance aspects (Woerner et al., 2019; Bakdi et al., 2021) and situational awareness aspects (Vasstein, 2021). Some work is also emerging on formal methods for design and verification of autonomous vessels (Shokri-Manninen et al., 2020; Foster et al., 2020). We strongly believe that design and verification of autonomous vessels needs to be modular in order to manage the

* Corresponding author.
E-mail address: tobias.torben@ntnu.no (T.R. Torben).
URL: https://www.ntnu.edu/employees/tobias.torben (T.R. Torben).

https://doi.org/10.1016/j.oceaneng.2023.113685
Received 15 October 2022; Received in revised form 18 December 2022; Accepted 10 January 2023
Available online 17 January 2023

immense complexity. The current research frontier addresses methodology for system-level risk assessment and requirement derivation as well as methodology to verify certain modules of autonomous vessels. However, we see an unresolved need for a modular methodology to go from system-level requirements and verification to detailed module requirements and verification in a coherent and coordinated way. We propose a contract-based approach as a candidate to address these needs.

Contract-based design is a modular approach to system design. The system is modularized into encapsulated design units called *components*. Each component is associated with an assume-guarantee contract which specifies what the component assumes about its environment and what behavior it can guarantee given that these assumptions hold. Contract-based design can also support stepwise refinement, where a high-level contract for a component at a high abstraction level is incrementally refined into more detailed contracts for sub-components at lower abstraction levels (Abrial, 2011). The ultimate goal of taking a contract-based design approach is to enable compositional reasoning, that is, reasoning about the correctness of a composed system based on the contracts of individual components. Compositional reasoning includes both checking that the contracts of connected components are compatible and that the composition of contracts for sub-components correctly refines the higher-level contract. By taking on a suitable mathematical formalism when specifying the contracts, it is also possible to reason about the correctness of a composition or refinement step in a formal, mathematical manner (Cimatti and Tonetta, 2012). The use of contracts for specification was originally introduced in the context of software engineering, with the *design by contract* methodology of Meyer (1992). In the same period, the foundation for compositional reasoning was also developed in the pioneering work of Clarke et al. (1989) as a divide-and-conquer solution to the scalability issues of model checkers. Over the past two decades, significant research has attempted to apply contract-based methods to cyber–physical systems resulting in more complete frameworks combining specification, refinement, and verification (Sangiovanni-Vincentelli et al., 2012; Nuzzo et al., 2015). Contract-based methods have seen application in other industries which are faced with complex and safety critical systems, such as aviation (Nuzzo et al., 2014) and automotive (Benveniste et al., 2008). In the maritime industry, however, contract-based methodologies have not yet seen any adoption. A notable exception is the recent publication of Hake et al. (2021), which proposes a contract-based methodology for verifying software updates on shipboard equipment. The authors are not aware of any previous research targeting contract-based verification for autonomous vessels.

We believe that taking a contract-based design approach has the potential to address many of the challenges related to design and verification of autonomous vessel control systems. Having modularity in a design is a well-proven technique for decomposing and managing complexity. Modularity often also comes as a necessity in maritime control systems when integrating custom and commercial-off-the-shelf (COTS) components from several different vendors. Having a more structured and formalized integration of control systems and equipment from different vendors is a long-standing need in the maritime industry (Smogeli et al., 2020). Another challenge for complex systems in general, and autonomous systems in particular, is to derive a complete and coherent set of system requirements to ensure safe and correct behavior (Rokseth and Utne, 2019). In a contract-based setting, the contracts act as the requirements for individual components. Because components can be structured hierarchically, meaning that a component can be implemented by a set of sub-components, we can analyze the system at different abstraction levels. This may simplify the derivation of requirements, as the derivation of detailed sub-components contracts often emerges naturally as a combination of refining the higher-level contract and being compatible with other sub-component contracts. Refinement checking ensures the completeness of contracts with respect to the top-level contract. Another benefit of deriving assume-guarantee contracts is that all assumptions in the system are made explicit and may be monitored online as an additional safety function. In addition to compositional reasoning about contracts, a central step in the verification process is to show that each component complies with its contract. Due to the complexity and the types of algorithms used in autonomous vessel control systems, formal verification of contract compliance will likely be impossible with the current state-of-the-art. Therefore, we propose simulation-based testing as an alternative solution for verifying contract compliance (Pedersen et al., 2020). Although exhaustive verification cannot be achieved with a testing method, simulation-based testing already has a strong and successful history in the verification of maritime control systems (Smogeli and Skogdalen, 2011; Smogeli, 2015), and several methodologies exist for increasing the exhaustiveness (Torben et al., 2022a). It will be beneficial to use a combination of several different simulators to achieve both realistic and integrated simulations and high test coverage. We believe that a contract-based framework can be used together with simulation-based testing in a mutually beneficial way. The simulation-based testing acts as a means to generate evidence of contract compliance, and the contract framework ensures that the testing efforts combine in a structured and coordinated way in order to build evidence of the overall system correctness.

The research objective of the current work is to investigate if a contract-based design approach can enable more structured and formalized verification of autonomous vessel control systems, and the main scientific contribution is a top-down methodology for contract-based verification of such systems. This paper is outlined as follows. In Section 2 we first introduce some mathematical foundations before we present a framework for defining components and specifying contracts using the Z3 theorem prover (de Moura and Bjørner, 2008). In Section 3 we present a methodology for contract-based verification using the framework of Section 2. The methodology is divided into 4 steps: (1) Hazard identification between the autonomous vessel and the operative environment in order to define the top-level component and contract, (2) stepwise refinement of the top-level component into detailed sub-components and contracts, (3) definition of test setups for simulation-based testing to verify that components meet their contract, and (4) applying a recursive procedure for contracts-based system verification. In Section 4, we demonstrate the use of the contract-based framework and methodology in a case study with an autonomous passenger ferry. In Section 5 we discuss some challenges of the methodology and possible approaches to address these. Concluding remarks and suggestions for future work are given in Section 6.

2. Contract framework for system verification

In this section, we introduce the contract framework used in this work. We begin by presenting the mathematical foundations, where components and contracts are abstractly defined in terms of sets of behaviors. Then we go on to introduce the Z3 theorem prover and show concretely how components and contracts can be specified using the syntax of Z3. Table 1 is included to give an overview of the most used symbols.

2.1. Preliminaries: Mathematical foundation for contract-based design

The mathematical foundation for the framework is defined using set notation. This is based on the assume-guarantee contract theory from Benveniste et al. (2018), to which the reader is referred for further details.

Consider the variables $v_1, v_2, \ldots, v_n$, each with domain $D_1, D_2, .., D_n$. Let $V := D_1 \times D_2 \times \cdots \times D_n$. A *reaction* $s \in V$ is defined as a vector assigning values from V to each variable. A *behavior* σ is defined as a, possibly infinite, discrete-time sequence of reactions $\sigma = s_1, s_2, s_3, \ldots$. We add a special symbol, $\perp$, to the domain of each variable, to indicate the absence of a reaction to a particular variable and define the

Table 1
Overview of the most used symbols in this paper.

σ	Behavior
M	Component
C	Contract
A	The set of behaviors that define the assumptions of an assume-guarantee contract
G	The set of behaviors that define the guarantee of an assume-guarantee contract
φ_A	The logic formula which specifies the assumptions A of a contract
φ_G	The logic formula which specifies the guarantees G of a contract
b_C	Observer for the contract C

silent reaction ϵ as the reaction which assigns $\perp$ to each variable. For simplicity, we consider only synchronous behaviors here, where each variable is assigned a reaction simultaneously at discrete time steps. Extensions to consider asynchronous behaviors may be achieved using Kahn Process Networks (Kahn, 1974), as proposed by Benveniste et al. (2018).

A *component*, M, is described abstractly in terms of the set P of behaviors the component exhibits. In practice, the set of behaviors for a component can for instance be specified in terms of a differential equation or a computer program. The *composition* of two components $M_1 \times M_2$ is defined as the intersection of their respective sets of behaviors $P_1 \cap P_2$.

A *contract* is defined as a pair of assertions $C = (A, G)$, where A are called the assumptions and G are called the guarantees. The set $\mathcal{E}_C$ of legal environments is the collection of all components E for C, such that $E \subseteq A$. The set $\mathcal{M}_C$ of legal implementations of C is defined by the collection of components M such that $A \times M \subseteq G$. All contracts which admit the same set of behaviors are by definition equivalent. A contract is *saturated* if $G = G \cup A^c$, where A^c is the complement of A. Contract saturation means that the set of guarantees is maximal in the sense that it contains all behaviors where the assumptions do not hold. Since the guarantees only are in force when the assumptions hold, all contracts can be transformed into an equivalent saturated contract.

Next, we define a set of contract operations. Let $C_1 = (A_1, G_1)$ and $C_2 = (A_2, G_2)$ be two saturated contracts defined over the same set of variables. The refinement relation $C_2 \leq C_1$ is defined by

$$C_2 \leq C_1 \text{ iff } G_2 \subseteq G_1 \text{ and } A_2 \supseteq A_1.$$

If this relation is satisfied, we say that C_2 refines C_1. Refinement is an ordering of the relative strength of contracts. Informally, a contract has to have as strong or stronger guarantees and as permissive or more permissive assumptions as another contract in order to refine it.

The *conjunction* of C_1 and C_2, denoted $C_1 \wedge C_2$, is defined as

$$C_1 \wedge C_2 := (A_1 \cup A_2, G_1 \cap G_2).$$

Typically, conjunction is used to impose several different contracts on a component, such that the component needs to comply with each of them in order to comply with the conjunction.

The *composition* of two contracts is denoted $C_1 \otimes C_2$. Composition is defined as

$$C_1 \otimes C_2 := (G_1 \cap G_2, (A_1 \cap A_2) \cup (G_1 \cap G_2)^c)$$

Similar to the composition of components, composition of contracts can be used to construct composite contracts out of simpler ones. For instance, if a component is implemented by a set of sub-components, we may want to verify that the composition of the contracts for the sub-components refines the contract of the parent component. From the definition of composition, it can be seen that a component has to comply with the guarantees of both C_1 and C_2. However, the assumptions of the composite contract are relaxed, as some of the assumptions may be covered by the guarantees of the other contract. It can be shown that using these definitions, the conjunction and composition operators are associative and commutative.

Finally, we define the concept of *observers*. Observers are in this context used for evaluating whether a single behavior complies with a contract or not, which is central when testing a component. For a contract $C = (A, G)$ and a behavior σ, the observer $b_C(\sigma, A, G) = True$ iff $\sigma \in G \cup A^c$. Otherwise, $b_C(\sigma, A, G) = False$.

2.2. Contract framework using the Z3 theorem prover

We continue by presenting a concrete contract framework that uses the syntax of the Z3 automated theorem prover to specify components and contracts and show how this relates to the abstract framework of Section 2.1.

2.2.1. A short intro to Z3

Z3 is an open-source automated theorem prover (ATP) developed by Microsoft Research (de Moura and Bjørner, 2008). ATPs are tools that are given a mathematical theorem as input and attempt to automatically prove or disprove it. Z3 is an ATP of the Satisfiability Modulo Theories (SMT) solver type. To understand SMT solvers, we first introduce the simpler and more well-known boolean satisfiability solvers, commonly referred to as SAT solvers. SAT solvers attempt to decide if a boolean expression in the form of a propositional logic formula is satisfiable or not, that is, if there is a combination of true or false assignments to the boolean variable which makes the formula true. SAT solving is nondeterministic polynomial-time complete (NP-complete), and a wide range of important NP-complete problems can be rephrased as SAT problems. Significant effort has therefore been put into developing efficient heuristics for SAT solving, such that state-of-the-art SAT solvers can solve problem instances involving several hundred thousand variables.

SMT solvers generalize SAT solvers from boolean formulas to more complex formulas involving e.g. integers, real numbers, arrays, or strings. This is achieved by combining a SAT solver with a set of domain-specific theories, such as linear arithmetic. SMT solvers can be used as ATPs by checking the satisfiability of the negation of a theorem. If the negation of the theorem is found to be unsatisfiable, then the theorem is proved. If, on the other hand, a solution is found which satisfies the negation of the theorem, this will disprove the theorem, and the SMT will return the solution as a counterexample.

Z3 operates on first-order logic formulas, which in addition to the operators of propositional logic (And ($\wedge$), Or ($\vee$), Negation ($\neg$) and Implication ($\rightarrow$)), contain the universal quantifier "for all", denoted by the symbol $\forall$, and the existential quantifier "there exists", denoted by the symbol $\exists$. Z3 has bindings for several programming languages. In this paper we will use the Python bindings as the syntax to define components and contracts due to its good readability and Python being a widely known language.

2.2.2. Component model

Next we define the component model of our contract framework. The model is illustrated conceptually in Fig. 1. The component interface consists of *out-ports*, which are controlled by the component, and *in-ports*, which are controlled by the environment of the component. In addition, a component can have a set of *parameters* which are constant during a simulation. The communication between components is achieved by sending *messages* on out-ports that are received on the in-ports of other components. The data structure of a message is defined by its *message type*. The message type is defined by declaring a set of variables defined by a variable name and data type. Variables can be nested to form struct-like data structures. The allowed data types are the Z3 basic data types *Real, Int* and *Bool*. The basic data types also have fixed-length vectorial versions, denoted *RealVector, IntVector* and

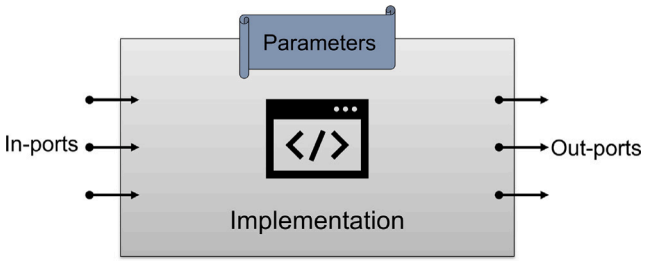

Fig. 1. Conceptual view of the component model used in this work.

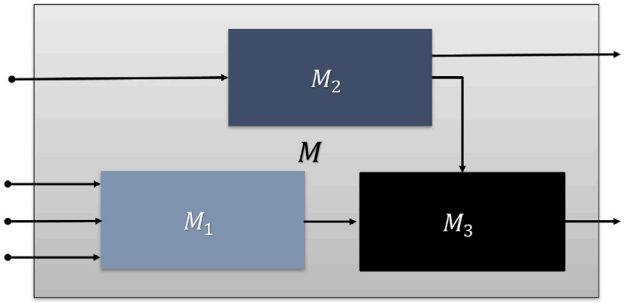

Fig. 2. Example of a composite component structure with two abstraction levels. Component M is implemented by the sub-components M_1, M_2, and M_3 such that $M = M_1 \times M_2 \times M_3$.

BoolVector. We note that Z3 supports several other datatypes, such as strings, arrays, bit vectors and even floating point numbers, however, this is not explored further in our work. For examples of message type definitions, the reader is referred to the case study in Section 4.2.

The *implementation* of a component synchronously reads messages from the in-ports and assigns messages to the out-ports at each discrete time step. Assignment of a *None* object is used to indicate the absence of an assignment (similar to the silent reaction in Section 2.1). We separate between two types of components: Composite components, which are implemented by composition of sub-components, and atomic components which are implemented directly, for instance as a C program or a ROS node. The fact that a component can be implemented by a set of sub-components is a key concept in our component model. This means that components can be structured hierarchically, which enables a system to be analyzed at different *abstraction levels*. This is an effective approach to manage the complexity both during the design and verification phases. The following section will show how we propose to utilize the hierarchical component structure in our verification framework. An example of a composite component is given in Fig. 2.

To draw a comparison back to the abstract framework of Section 2.1, the variables $v_1, v_2, \ldots, v_n$ correspond to the in-ports and out-ports of the component. The domains of each variable, $D_1, D_2, .., D_n$ are defined by the corresponding message type of the component. A behavior is a discrete sequence of messages assigned to each port of the behavior, and the set of behaviors for the component is implicitly defined by the implementation of the component. Composition of components is achieved by connecting the out-ports of one component with the in-ports of another component with the same message type.

2.2.3. Contract specification in Z3

As introduced in Section 2.1, contracts characterize the legal environments and legal implementations for components. In assume/guarantee contracts, a contract C is defined by a pair of assertions that specify the set of legal environment behaviors, called the assumptions A, and the set of guaranteed component behaviors given that the assumptions hold, called the guarantees G. We propose to use first-order logic formulas in the Z3 syntax to specify the sets A and G of the contract.

In our framework, a contract is linked to a specific component and therefore acts as a specification of requirements for the component. To preserve modularity, a contract may only refer to the ports of the component it is linked to. A contract consists of two first-order logic formulas φ_A and φ_G which define the assumptions and guarantees, respectively. The basic building blocks of the logic formulas are *predicates*, which are functions $\pi : V \mapsto \mathbb{B}$ mapping values of the variables v to a boolean value. An example of a predicate on the variable $x \in \mathbb{R}$ is $x \leq 10$. In the formulas φ_A and φ_G, several predicates are combined using the first-order logic operators. $x \leq 10 \wedge x \geq 0$ is an example of a formula with two predicates connected by the $\wedge$ operator. Usually, we wish to specify several assumptions and guarantees for a component. This can easily be achieved by letting the formulas φ_A and φ_G be formulated as a conjunction of individual assumptions and guarantees.

Comparing this with the abstract definition of assume/guarantee contracts in Section 2.1, the sets A and G are now defined by the logic formulas φ_A and φ_G. We only consider specification of time-invariant properties in this work, that is, properties that must hold at each time step. Extensions to consider temporal properties may be achieved by using a temporal logic, such as Signal Temporal Logic (STL) (Maler and Nickovic, 2004). For time-invariant assumptions and guarantees, the sets A and G are simply subsets of V. The formulas (φ_A, φ_G), and the sets (A, G) are related as follows. Each predicate π_i in the logic formulas defines a set $\mathcal{O}(\pi_i) \subseteq V$ where that predicate is true. When combining predicates into a logic formula, the set that the formula defines is defined for each logical operation. Logical conjunction, $\pi_1 \wedge \pi_2$ translates to set intersection $\mathcal{O}(\pi_1) \cap \mathcal{O}(\pi_2)$. Logical disjunction, $\pi_1 \vee \pi_2$ translates to set union $\mathcal{O}(\pi_1) \cup \mathcal{O}(\pi_2)$. Logical negation, $\neg\pi_1$ translates to set complement $\mathcal{O}(\pi_1)^c$. Translation of the logical quantifiers ($\forall$ and $\exists$) is also possible, but this is not explored further in our work.

As shown in Section 2.1, a behavior σ complies with the contract if $\sigma \in G \cup A^c$. In terms of the first-order logic formulas, σ complies with the contract iff $\varphi_G(\sigma) \vee \neg\varphi_A(\sigma) = True$. This coincides the definition of logical implication: $p \rightarrow q := q \vee \neg p$. Hence, a behavior σ satisfies the contract $C = (\varphi_G, \varphi_G)$ iff $\varphi_A(\sigma) \rightarrow \varphi_G(\sigma) = True$. Moreover, this entails that the observer for C is simply defined as $b_C(\sigma, \varphi_A, \varphi_G) = \varphi_A(\sigma) \rightarrow \varphi_G(\sigma)$. For specific examples of contracts, the reader is referred to Section 4.2 of the case study.

3. Methodology for contract-based verification of autonomous vessels

In this section, we propose a step-by-step methodology for contract-based verification of autonomous vessels using the framework introduced in Section 2. Step 1 defines the top-level level component and specifies the top-level contract in order to ensure safe interactions with the operative environment. Step 2 refines the top-level components and contracts in a series of refinement steps until we reach a sufficient level of detail. Step 3 defines the simulation-based test setup for each component. Finally, Step 4 applies a recursive algorithm that verifies contract refinement and runs simulation-based testing to verify contract compliance. The methodology is illustrated in Fig. 3. We give a few illustrative examples in this section. Instead, we recommend the reader to look at the corresponding steps in the case study of Section 4 for concrete examples.

Step 1: Define the top-level component and contract

To enable contract-based verification for an autonomous vessel, we must first define the component structure and assign contracts to the components. This can be done both during the design of a new autonomous vessel and by formulating an existing autonomous vessel design in terms of components and contracts. In Step 1, we begin at

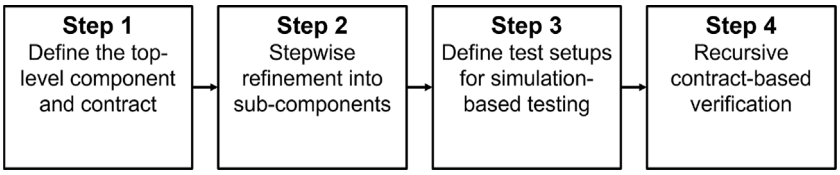

Fig. 3. The 4 steps of the proposed methodology for contract-based verification.

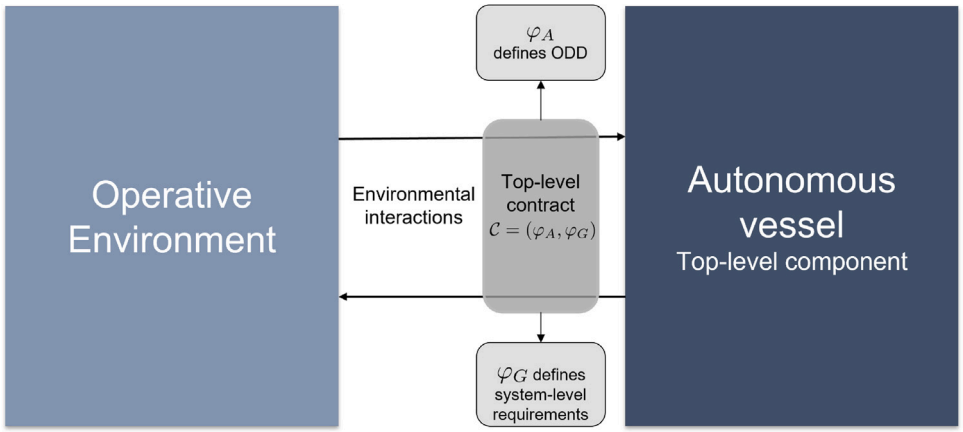

Fig. 4. Step 1: Define the top-level component and contract.

the top-level view, shown in Fig. 4. This consists of two components; the autonomous vessel and its operative environment. The first step is modeling the operative environment and identifying all relevant interactions between the autonomous vessel and its operative environment. This involves creating the necessary port definitions and corresponding message types between the autonomous vessel component and the operative environment component.

Next, the top-level contract between these components must be formulated. The assumptions of the top-level contract specify which environmental conditions the autonomous vessel is designed to operate under, and thus define the operational design domain (ODD). The guarantees of the top-level contract form the system-level requirements on the behavior of the autonomous vessel.

Achieving completeness both in identifying relevant interactions with the operative environment and constraining these in the top-level contract is of paramount importance in the verification process. Achieving sufficient completeness in requirements and identifying relevant test scenarios for verification are generally major challenges in the assurance process for complex systems (Rokseth and Utne, 2019) and in particular for systems with a high level of autonomy (Rokseth et al., 2019). The STPA methodology provides a systematic top-down approach to hazard identification and generation of loss scenarios which may constitute an important foundation for increasing the completeness when identifying interactions and formulating the top-level contract. Moreover, we believe that our top-down approach simplifies the requirement identification process since the top-level contract is formulated at a high level of abstraction. More detailed requirements are either derived from the top-level contract or emerge at lower abstraction levels to constrain the interactions between sub-components. This allows us to focus on specifying the desired system-level behavior at this stage, such as keeping a minimum distance to other vessels, and leave the details regarding how these requirements are satisfied and all the ways in which they can be violated, to the more detailed requirements further down in the abstraction levels.

Note that the assumptions of the top-level component are special. Since they are assumptions on the operative environment, they are outside of our control. Examples of environmental assumptions are assuming that the speed of other vessels is below some threshold, or that the weather conditions satisfy certain criteria. Even though we cannot control the environment to comply with these assumptions, we believe it is important to specify them. Having awareness of the assumptions of the environment is critical to designing a safe and robust system, and it allows us to monitor these assumptions and take appropriate action if they are violated.

Step 2: Stepwise refinement into sub-components

After the top-level component and contract have been established in Step 1, Step 2 involves a series of design iterations where the components and contracts are incrementally refined into a more detailed and implementation-ready form. This process takes place as a series of well-defined refinement steps. In each refinement step, a component is decomposed into a set of sub-components, as illustrated in Fig. 2. Additionally, contracts must be assigned to each new sub-component. The composition of the contracts of the sub-components has to refine the contract of the component which the sub-components implement. The contracts of the sub-components also need to be compatible with each other, that is, if an out-port of component M_i with contract $C_i = (\varphi^i_A, \varphi^i_G)$ is connected to an in-port of component M_j with contract $C_j = (\varphi^j_A, \varphi^j_G)$, then the guarantees of C_i have to be strong enough to enforce the assumptions of C_j. Note that the actual checking of the correctness of the refinement is done in Step 4.

Arriving at component architecture with corresponding contracts that respect both refinement and compatibility is a design exercise that requires good engineering judgment. Architectural decisions must be made, custom components must be designed and COTS components must be selected. This is an iterative process that often involves contract negotiations. For instance, when composing a set of sub-components, one may realize that the selected sensor does not have the precision required to refine the contract of the parent component. In this case, a new design iteration is required, for instance by selecting a more precise sensor or designing a more precise controller. In fact, this is very similar to the types of decisions that engineers typically make during system design, however, the contract-based concepts of contracts, composition, and refinement give a language and framework to analyze these questions in a more structured and formalized manner (Benveniste et al., 2018).

The refinement process of Step 2 should be continued until all components are described at a level of detail that can readily be implemented in hardware or software. Moreover, all components should be decomposed until they are simple enough to be verified to the desired level of assurance. For critical components, the refinement may continue all the way to individual software functions, whereas the refinement of less critical components may be stopped at a higher level of abstraction. In some cases, the refinement stops naturally at COTS components.

Step 3: Defining test setups for simulation-based testing

Having the full system structure, in form of a component structure with corresponding contracts, Step 3 involves defining test setups for simulation-based testing of the components. We first give some background on why we propose to combine simulation-based testing and contract-based verification before we state the specific activities for Step 3.

3.1. Combining simulation-based testing and contract-based verification

Testing is a method for verifying a component where only a selected subset of the behaviors of the component is checked. This is the most common means of verification for embedded systems, because exhaustive verification is often too time-consuming or not feasible at all (Kapinski et al., 2016). In *simulation-based* testing, the component is tested in a simulated environment. Parts of the component itself may also be simulated in order to focus testing on certain aspects. Some common component representations for simulation-based testing are Hardware-in-the-Loop (HiL), where the real component software runs on the real component hardware, Software-in-the-Loop (SiL), where the real component software runs on virtualized hardware, and Model-in-the-Loop (MiL), where a simulation model of the component is used (Torben et al., 2022a). Simulation is able to analyze the system-level behavior of highly complex systems. Autonomous vessels are characterized by high levels of complexity in their hardware and software systems, as well as in their complex interaction with the operative environment. Combined with the intrinsic challenges related to verification of autonomous functions, such as the use of machine learning components and hard-to-predict emergent behaviors, currently there exist no viable alternatives to simulation-based testing for verifying their system-level behavior. It is widely agreed upon that simulation-based testing will be a key solution for the assurance of autonomous vessels (Pedersen et al., 2020). Next, we show how existing methodologies for simulation-based testing and contract-based verification can be combined in a mutually beneficial way.

A central activity in contract-based verification is to prove that a component complies with its contract. We call this activity *contract compliance checking*. Most previous examples of contract-based verification have used formal verification techniques for contract compliance checking, such as model checking (Clarke, 1997). This is attractive, as it results in a rigorous mathematical proof of contract compliance. However, due to the overall complexity, the hybrid system dynamics, and the use of advanced control techniques such as machine learning algorithms and model predictive control (MPC), state-of-the-art formal verification techniques are not capable of formally verifying all aspects of autonomous vessel control systems. Thus, we propose simulation-based testing as an alternative approach to contract compliance checking. Instead of formally verifying that all behaviors of a component comply with the contract of the component, simulation-based testing will generate evidence of contract compliance by running an adequate number of simulations in well-chosen scenarios and evaluating the resulting behaviors against the contract using an observer for the contract. Existing methodology for simulation-based testing, which addresses scenario selection and coverage assessment can readily be used for contract compliance checking.

Hence, simulation-based testing may solve a problem for contract-based verification of complex autonomous systems. At the same time, contract-based verification addresses known challenges for simulation-based testing. High-fidelity simulation of an entire autonomous vessel and its operative environment is possible using 3D-rendering engines for exteroceptive sensor simulation and software-in-the-loop for including exact replicas of the control software in the simulations. However, such simulations are typically highly computationally expensive resulting in low simulation speed and corresponding limited test coverage. Many sub-systems can be simulated accurately using simplified models that are orders of magnitude less computationally expensive, and thus achieve high test coverage. This suggests that several different simulators should be combined in order to address both system-level behavior at a high level of integration and sub-system behavior with high test coverage. However, using several different simulators raises the question of how the testing efforts by each of these should be combined in a coordinated manner in order to build verification evidence for the autonomous vessel. We believe that contract-based verification addresses this question. The component structure of the system specifies the simulator taxonomy directly since each component will need its own simulator to perform the contract compliance checking. The contracts between the components ensure that the simulation efforts combine in a structured and coherent way towards achieving an overall assurance of the autonomous vessel. Additionally, as shown in the following, contract-based verification provides a good framework for test management.

3.2. Activities for Step 3

Step 3 of the methodology involves defining the *test setup* for each component in the system. A schematic view of a generic test setup is shown in Fig. 5. We split the test setup into two parts; the simulator and the test management system. The simulator part consists of the component under test in a HiL, SiL, or MiL representation, connected with a simulation model of the environment of the component, which generates the test inputs to the component under test. The simulator is connected to the test management system over a test interface, which outputs a behavior, σ, for each simulation. The behavior is evaluated against the contract $C = (\varphi_A, \varphi_G)$ of the component using the observer $b_C(\sigma, \varphi_A, \varphi_G)$, as described in Section 2.2.3. The assumptions φ_A of the contract for the component under test should also be used to define and focus the test scenarios. The scenario selection may use the evaluation of previous simulations to adaptively select new scenarios. This can for instance be achieved by guiding the scenario selections towards regions of the scenario space with poor performance or high uncertainty, as proposed in Torben et al. (2022a).

Step 4: Recursive contract-based verification

Having defined the component structure, assigned contracts, and created a taxonomy of test setups for simulation-based contract compliance checking, the system is finally ready for contract-based system verification. The goal of this step is to prove that all refinement steps from the top-level contract down to the atomic component contracts are correct, that the contracts of all connected components are compatible, and that all components comply with their contract. If this is achieved, we have produced substantial verification evidence to show that the implementation of the system meets the top-level contract. If the top-level contract is sufficiently complete, this ensures that the autonomous vessel exhibits safe and correct interactions with its operative environment. If the verification fails at some stage, a design change is required.

We split the contract-based verification procedure into three main activities, which will be applied recursively down through the component structure:

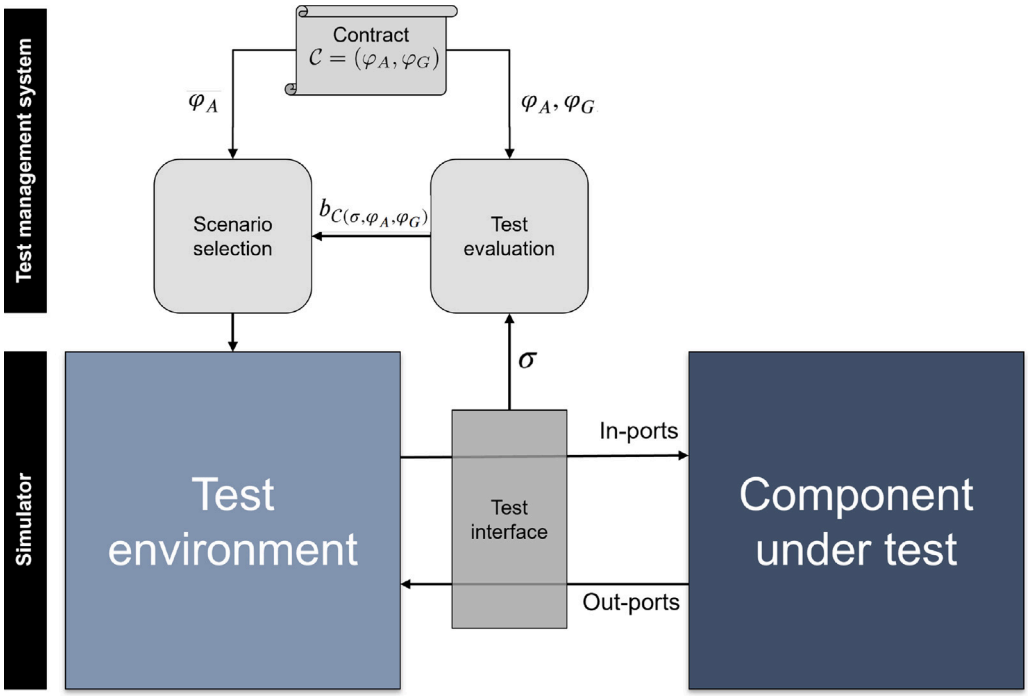

Fig. 5. Step 3: Test setup for simulation-based contract compliance checking.

Fig. 6. The milliAmpere II ferry docking at the Ravnkloa side of the Trondheim canal. Photo: Egil Eide.

- Contract compliance checking
- Contract composition
- Refinement checking

We first show the derivation of how to perform these activities using the contract framework of Section 2. Then, we present the recursive procedure to be applied in Step 4, which combines these activities.

3.3. Derivation of contract-based verification activities

Contract compliance checking has already been briefly introduced in Section 3. This involves verifying that a component complies with its contract by simulation-based testing with the test setup defined in Step 3. Stated more formally, contract compliance checking involves verifying that all behaviors σ of a component with contract $\mathcal{C} = (\varphi_A, \varphi_G)$ satisfies $\varphi_A(\sigma) \rightarrow \varphi_G(\sigma)$.

Contract composition involves composing the contracts for all sub-components into one composite contract. Suppose the component M with contract $C = (\varphi_A, \varphi_G)$ is implemented by k sub-components $M_1, M_2, \ldots, M_k$ with contracts $C_1, C_2, \ldots, C_k$, such that $M = M_1 \times M_2 \times \cdots \times M_k$. Recall that the composition of two saturated assume-guarantee contracts $C_i = (A_i.G_i)$ and $C_j = (A_j.G_j)$ is defined in terms of sets of behaviors as $C_i \otimes C_j = (A_{C_i \otimes C_j}, G_{C_i \otimes C_j})$, with

$$A_{C_i \otimes C_j} = (A_i \cap A_j) \cup (G_i \cap G_j)^c$$

$$G_{C_i \otimes C_j} = G_i \cap G_j.$$

Transforming this from set-notation to first-order logic formulas using the translations given in Section 2.2.3 yields the composite contract

$$\varphi_A^{C_i \otimes C_j} = (\varphi_A^i \wedge \varphi_A^j) \vee \neg(\varphi_G^i \wedge \varphi_G^j)$$

$$\varphi_G^{C_i \otimes C_j} = \varphi_G^i \wedge \varphi_G^j.$$

Also, recall that contract composition is associative and commutative, that is, $C_1 \otimes (C_2 \otimes C_3) = (C_1 \otimes C_2) \otimes C_3$ and $C_1 \otimes C_2 = C_2 \otimes C_1$. This means that we can construct the logic formula for the composite of any number of contracts using the above formulas by starting with C_1 and incrementally composing it with new contracts. For k contracts, the composite contract $C_{comp} = (\varphi_A^{comp}, \varphi_G^{comp})$ is thus constructed as $C_{comp} = (\ldots(((C_1 \otimes C_2) \otimes C_3) \otimes C_4)\ldots) \otimes C_k$.

Refinement checking involves checking that the composite contract, C_{comp}, for a set of sub-components refine the contract of the component which they implement. In mathematical terms, this involves checking that

$$C_{comp} = C_1 \otimes C_2 \otimes \cdots \otimes C_k \leq C.$$

We propose to formulate refinement checking as a theorem proving problem and feed it to the Z3 ATP. Recall that $C_2 \leq C_1$, iff $A_1 \subseteq A_2$ and $G_2 \subseteq G_1$. Translating this to first-order logic formulas and substituting the composite contract C_{comp} for C_1 and the contract of the parent component C for C_2 yields the following two theorems which must be proved for refinement checking

$$\varphi_A \rightarrow \varphi_A^{comp}$$

$$\varphi_G^{comp} \rightarrow \varphi_G$$

Although it may not be obvious, this method of refinement checking also checks that the contracts of sub-components are compatible with each other. We will briefly demonstrate why the refinement check will fail if there are incompatible contracts among the sub-components. When the pair contracts (C_i, C_j) are not compatible, then the assumptions of the composite contract, φ_A^{comp} will have assumptions from φ_A^j which are not covered by φ_G^j. Moreover, these uncovered assumptions cannot be covered by the assumptions of the parent component, M, since they are part of an internal port connection between sub-components and therefore not part of the interface of M. The refinement proof of $\varphi_A \rightarrow \varphi_A^{comp}$ will therefore fail, since there will be cases where φ_A holds but φ_A^{comp} does not.

3.4. The recursive contract-based verification procedure of Step 4

Next, we state the recursive procedure used in Step 4 which combines these three activities to achieve system verification. The procedure is shown in Algorithm 1.

We formulate the contract-based verification process as a recursive procedure, *verifyComponent()*. Step 4 simply involves applying this procedure to the top-level component, and it will recursively call itself for all sub-components. The recursion breaks when the procedure is applied to atomic components. The input to the procedure is a component M with contract $C = (\varphi_A, \varphi_G)$. The first step of the procedure is the contract compliance checking, which involves verifying that all behaviors σ of M satisfy $\varphi_A(\sigma) \rightarrow \varphi_G(\sigma)$ using methodology of choice. After this, the procedure branches depending on whether the component is composite or atomic. If it is atomic, the recursion breaks and the procedure terminates. If it is composite, the procedure proceeds to verify the sub-components which implement M. First, the contracts of all sub-components are saturated. Then, the composite contract $C_{comp} = (\varphi_A^{comp}, \varphi_G^{comp})$ of all sub-component contracts is incrementally constructed. Refinement checking is achieved by applying Z3's *prove* function to the theorems $\varphi_A \rightarrow \varphi_A^{comp}$ and $\varphi_G^{comp} \rightarrow \varphi_G$. Finally, the procedure recursively calls itself with each sub-component of M as the arguments. This will do the contract compliance check for all sub-components against their respective contracts, and continue to verify their respective sub-component structures if they are composite. Hence, by applying Algorithm 1 to the top-level component, the entire component tree of the system will be verified.

Algorithm 1: verifyComponent(Component M)

input: Component M with contract $C = (\varphi_A, \varphi_G)$
// Contract compliance checking
Verify that all behaviors σ of M satisfies $\varphi_A(\sigma) \rightarrow \varphi_G(\sigma)$ using methodology of choice;
// If the M is composite, verify the sub-component structure
if *M is composite* **then**
  // Saturate all sub-component contracts $C_1, C_2, \ldots, C_k$
  for $i = 1{:}k$ **do**
    $\varphi_G^i = \varphi_G^i \vee \neg\varphi_A^i$;
  end
  // Composition of the sub-component contracts $C_1, C_2, \ldots, C_k$
  $C_{comp} = (\varphi_A^{comp}, \varphi_G^{comp}) = C_1$;
  for $i = 2{:}k$ **do**
    $\varphi_A^{comp} = (\varphi_A^{comp} \wedge \varphi_A^i) \vee \neg(\varphi_G^{comp} \wedge \varphi_G^i)$;
    $\varphi_G^{comp} = \varphi_G^{comp} \wedge \varphi_G^i$;
  end
  // Refinement checking
  prove($\varphi_A \rightarrow \varphi_A^{comp}$);
  prove($\varphi_G^{comp} \rightarrow \varphi_G$);
  // Recursively apply verifyComponent() to each sub-component of M
  for *each sub-component M_i of M* **do**
    verifyComponent(M_i);
  end
end
return

4. Case study: The milliAmpere II autonomous passenger ferry

In this section, we demonstrate the use of our contract-based verification framework in a case study with the autonomous passenger ferry milliAmpere II.

4.1. Description of the vessel and operation

milliAmpere II is a small autonomous double-ended ferry designed for carrying 12 pedestrians and cyclists across the canal in Trondheim, Norway. The milliAmpere II is a follow-up on the research prototype milliAmpere (Brekke et al., 2022). In contrast to its predecessor, milliAmpere II will be put into regular passenger traffic and is therefore designed in accordance with the national regulations for passenger transport (NMD, 1990). The ferry is owned and will be operated by the Norwegian University of Science and Technology (NTNU) (see Fig. 6).

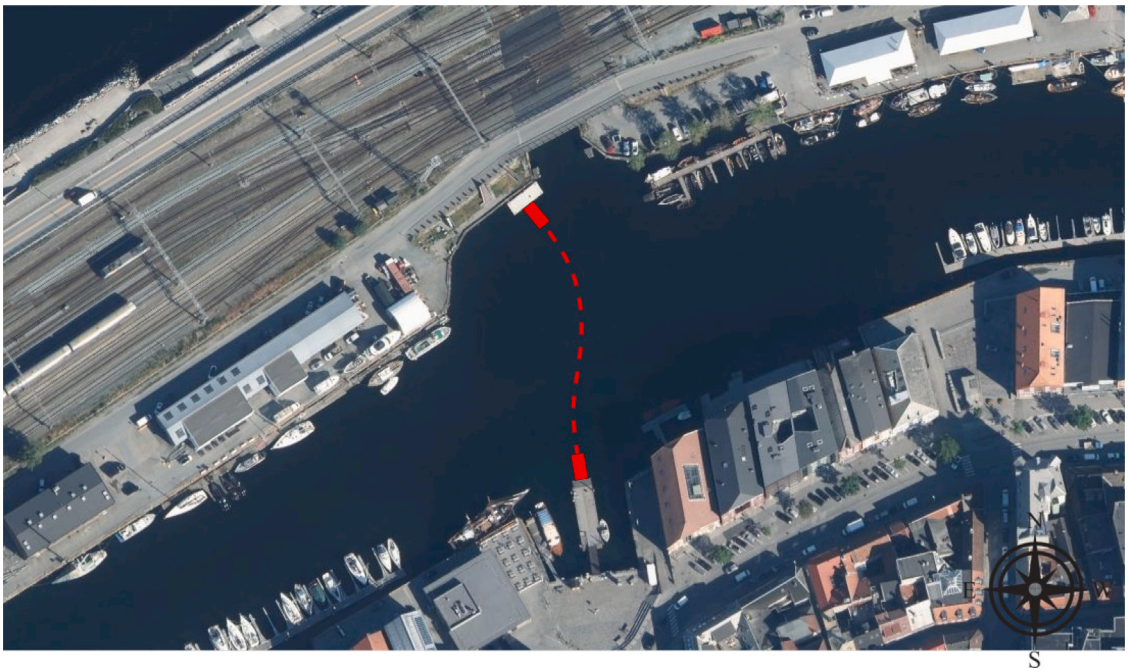

Fig. 7. The area of operation for milliAmpere II. The crossing is an 85 m long stretch between Ravnkloa and Vestre Kanalkai across the canal in Trondheim, Norway.

milliAmpere II has length 8.5 m and beam 3.5 m. It has a fully electric propulsion system with induction charging and four 10 kW thrusters with controllable azimuth angles. The service speed is 5 knots. The ferry is equipped with a class-approved motion control and automation system delivered by Marine Technologies. The navigation system of the ferry is a GNSS-aided inertial navigation system (INS) from SentiSystems with real-time kinematic (RTK) corrections. This system provides 6DOF motion measurements with centimeter-level positioning. The navigation system also features dead reckoning capabilities to ensure safe return to quay in case of a GNSS outage. The NTNU spin-off company Zeabuz has developed the high-level autonomy system. The ferry utilizes a range of different exteroceptive sensors for obtaining situational awareness. It is equipped with a top-mounted Simrad maritime radar, FLIR Boson infrared cameras, and FLIR Blackfly S RGB cameras, as well as Ouster OS1 lidars placed at the corners of the ferry. The autonomy system is responsible for processing the sensor data and for providing the motion control system with a collision-free motion reference. The ferry will travel along a fixed pre-planned path and only control the speed along that path to avoid collisions with other vessels. The collision avoidance algorithm is described in Thyri et al. (2020)

The area of operation is the stretch from Ravnkloa to Vestre Kanalkai. The crossing is approximately 85 m long, which is expected to take about one minute at normal service speed. The area is regulated as fully enclosed waters, with a speed limit of 5 knots (2.6 m/s). The maximum wave height in the area is 0.5 m and the maximum current speed is 3 knots (1.5 m/s). The area may be subject to harsh weather, and the ferry will be suspended if the wind speed exceeds 10 m/s. There are no shallows and no static obstacles in the area, apart from the sides of the canal and boats that are moored to the side of the canal. The traffic in the canal consists of motorized vessels of varying sizes and types. In addition, the canal is subject to heavy kayak and canoe traffic during the summer season. A bird's eye view of the area is given in Fig. 7.

4.2. Applying the contract-based verification methodology

Having introduced the object of the case study, we proceed to follow the steps in the contract-based verification methodology introduced in Section 3. The full component structure we develop in the case study is given in Fig. 8.

Step 1: Define the top-level component and contract

We start at the top-level view and define the interaction with the operative environment. As described in Section 3, this should be done by a systematic hazard identification process. To demonstrate the main features of the methodology in a brief and concise manner, we limit the scope to studying only a couple of key interactions with the operative environment. We consider the interactions with a moving obstacle and the environmental loads. We define an *obstacle_motion* out-port on the operative environment which carries messages of the *VesselMotionMsg* type:

```
class VesselMotionMsg:
    def __init__(self, name):
        self.speed = Real(name + '_speed')
        self.course = Real(name + '_course')
        self.position = RealVector(name +
                        '_position', 2)
```

The value on the *obstacle_motion* port will be *None* if there are no visible obstacles. We also define an in-port on the operative environment for the milliAmpere II motion, as the motion of milliAmpere II may influence the behavior of the obstacle. We name this port *ferry_motion* and it also carries messages of the *VesselMotionMsg* type. Finally, we define an out-port named *environment_loads* with message type *EnvLoadMsg* defined as:

```
class EnvLoadMsg:
    def __init__(self):
        self.current_speed =
        Real('current_speed')
        self.wind_speed =
        Real('wind_speed')
        self.wind_direction =
        Real('wind_direction')
        self.sig_waveheight =
        Real('sig_waveheight')
        self.peak_period =
        Real('peak_period')
```

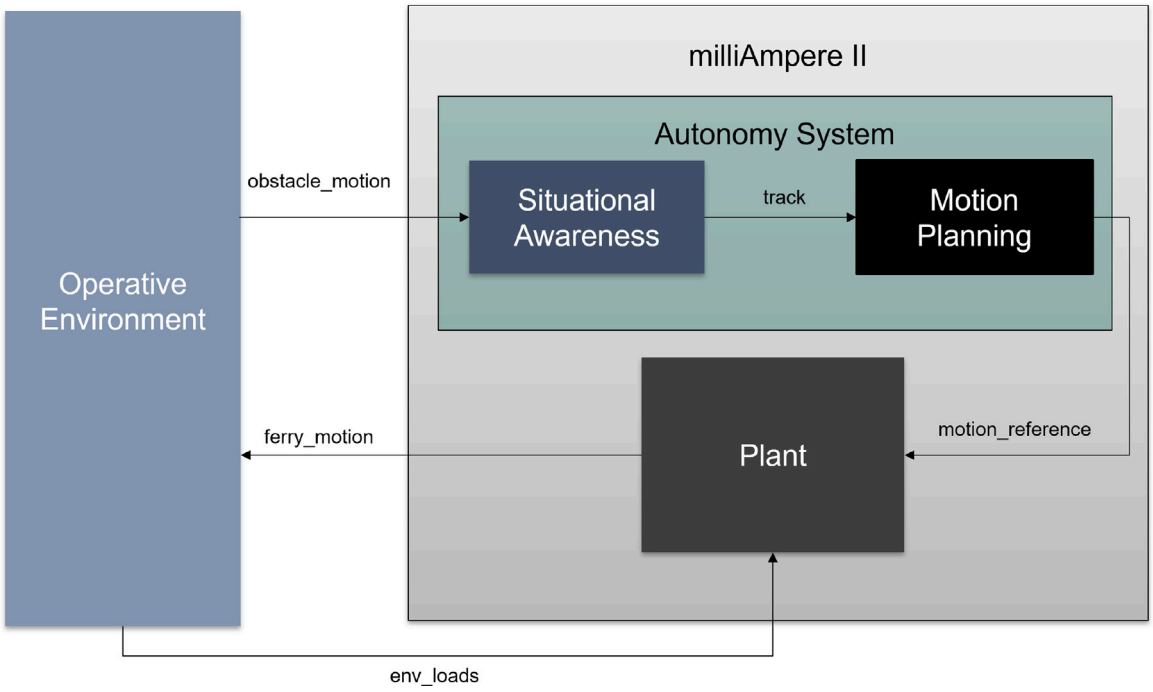

Fig. 8. The component structure used in the case study.

The current direction is assumed to be parallel to the canal, and the direction of flow is given by the sign of the current speed, where a positive speed represents downstream current (from west to east in Fig. 7).

Having defined the environmental interactions, the next step is to define the top-level contract. As Fig. 4 showed, this contract plays a fundamental role, as its assumptions form the ODD and the guarantees form the top-level requirements to ensure safe interactions with the operative environment. We define the following top-level contract:

```
mA2_assumptions =
    And(obstacle_motion.speed >= 0,
        obstacle_motion.speed <= 2.6,
        env_loads.current_speed >= -1.5,
        env_loads.current_speed <= 1.5,
        env_loads.wind_speed >= 0,
        env_loads.wind_speed <= 10.0,
        env_loads.wave_height >= 0,
        env_loads.wave_height <= 0.5)

mA2_guarantees =
    And(distance_2d(obstacle_motion.position,
    ferry_motion.position) >= 10.0,
    distance_2d(ferry_motion.position, path) <=
    1.0)
```

The assumptions are derived from the CONOPS for milliAmpere II. The assumptions specify that the obstacles keep the speed limit for fully enclosed waters of 5 knots (2.6 m/s), the current speed is less than 1.5 m/s, the wind speed is less than 10 m/s, and that the significant wave height is less than 0.5 m. For the guarantees of the top-level contract, we limit the focus to avoiding collisions with the moving obstacle and following the pre-planned path. The guarantees specify that the ferry's distance to the moving obstacle should be greater than 10 m and that the ferry does not deviate from the pre-planned path by more than 1 m. Since there are no static obstacles within these margins, these two guarantees ensure that there are no collisions. Other aspects could have been included here, such as passenger comfort by constraining the allowed speed and acceleration. Such aspects are excluded for the sake of brevity but would be incorporated in the contract in the same way. The Z3 *And* operator builds a logic formula which is the conjunction of all the arguments. The *path* variable is a 2-dimensional vector whose value is equal to the point on the pre-defined path which is closest to *ferry_motion.position* at all times. The *distance_2d* function is defined as:

```
def distance_2d(pos1,pos2):
    return Sqrt((pos1.north-pos2.north)**2
    + (pos1.east-pos2.east)**2)
```

Before we conclude Step 1, we refer back to Section 2 to show how these concrete component and contract definitions relate to the mathematical notation. At the top-level view, the variables $v_1, v_2, \ldots, v_{11}$ correspond to the 11 variables in the message type definitions of the *obstacle_motion, ferry_motion* and *env_loads* ports. That is, v_1 = *obstacle_motion.speed*, v_2 = *obstacle_motion.course*, ..., v_{11} = *env_loads.peak_period*. The numbering is chosen arbitrarily in this example. The domains of each variable, $D_1, D_2, .., D_{11}$ correspond to the data types of these variables, as defined in the message types. In this case, $D_1 = D_2 = \cdots = D_{11} = \mathbb{R}$, and $V = D_1 \times D_2 \times \cdots D_{11} = \mathbb{R}^{11}$. At each time step, the implementation of the components assigns values to each of these variables, resulting in a reaction $s \in \mathbb{R}^{11}$. The discrete-time sequence of reactions is the behavior, σ, at this abstraction level. Let $C_{mA2} = (\varphi_A, \varphi_G)$ denote the top-level contract. The *mA2_assumptions* specify the first-order logic formula φ_A and *mA2_guarantees* specifies φ_G. Each of these formulas is built by combining predicates over the variables $v_1, v_2, \ldots, v_{11}$ with first-order logic operators, in this case, the *And* operator.

Step 2: Stepwise refinement into sub-components

Having defined the top-level component and contract, in Step 2 we proceed by refining them into a set of sub-components that implement the top-level component. As Fig. 8 shows, in the first refinement steps we have chosen to split the top-level component (milliAmpere II) into two sub-components; the autonomy system and the plant, in a standard controller-plant configuration. The plant component contains the physical ferry and the low-level control functionality, such as the industrial motion control system, the actuator control, and the navigation system.

The interaction between the autonomy system and the plant takes place over the *motion_reference* port, where the autonomy system provides the plant with the trajectory that the motion control system should track. We define the *motion_reference* port to also send messages of the *VesselMotionMsg* type.

Next, we need to define the contracts at this abstraction level. The contracts need to refine the contract for the top-level component and be compatible across the two components. Additionally, the contracts may also need to consider any inherent limitations in the subcomponents, such as the precision of the motion control system. We begin by defining the contract for the plant. The assumptions on the environmental loads are simply copied from the top-level component. The assumptions on the motion reference constrain the speed reference to be between −2 m/s and 5 m/s, to receive a feasible trajectory to track. The guarantee of the plant specifies that the trajectory error of the motion control system should be less than 1 m.

```
plant_assumptions =
    And(env_loads.current_speed >= -1.5,
        env_loads.current_speed <= 1.5,
        env_loads.wind_speed >= 0,
        env_loads.wind_speed <= 10.0,
        env_loads.wave_height >= 0,
        env_loads.wave_height <= 0.5,
        motion_ref.speed >= -2.0,
        motion_ref.speed <= 5.0)

plant_guarantees =
    distance_2d(motion_ref.position,
    ferry_motion.position) <= 1.0
```

Next, we define the contracts for the autonomy system. The assumptions on the traffic port are copied directly from the top-level component. The guarantees specify that the motion reference needs to maintain a safe distance of 11 m from the obstacle. The safe distance from the ferry to the obstacle in the top-level contract was only 10 m, however, since the plant component can only track the motion reference from the autonomy system with a precision of 1 m, the safe distance guarantee for the motion reference needs to be increased to 11 m for the autonomy system and plant contracts to correctly refine the top-level contract. Additionally, we specify that the motion reference should coincide with the pre-planned path at all times, meaning that we only allow the autonomy system to generate references along this path, effectively meaning it is only controlling the ferry speed and position along the path. To be compatible with the plant contract, the autonomy system also needs to constrain the speed of the motion reference. The contract specifies that the speed reference will be between −1 m/s and 3 m/s, in order to keep well within the limitations in the plant assumptions.

```
autonomy_assumptions =
    And(obstacle_motion.speed >= 0.0,
        obstacle_motion.speed <= 2.6)

autonomy_guarantees =
    And(distance_2d(motion_ref.position, path)
        == 0,
        distance_2d(motion_ref.position,
        obstacle_motion.position) >= 11,
        motion_ref.speed >= -1.0,
        motion_ref.speed <= 3.0)
```

Finally, we will do one more refinement step on the autonomy system component. As Fig. 8 shows, we implement the autonomy system component by two sub-components; motion planning and situational awareness. The situational awareness component senses other vessels with its exteroceptive sensors and estimates their position, speed, and heading. This information is sent to the motion planning component over the *track* port. The motion planning component generates a *motion_reference* signal such that the ferry keeps a safe distance from the obstacle track.

The assumptions on the traffic port of the situational awareness component are copied from the autonomy system component. The guarantee of the situational awareness component specifies that if there is a visible obstacle, it will be detected and tracked with an error of less than 3 m:

```
sitaw_assumptions =
    And(obstacle_motion.speed >= 0,
        obstacle_motion.speed <= 2.6)

sitaw_guarantees =
    Implies(obstacle_motion is not None,
            distance_2d(track.position,
    obstacle_motion.position) <= 3.0)
```

For the *motion_planning* component, we define no assumptions on the *track* port. The guarantees specify that the motion reference should maintain a safe distance to the obstacle track received by the situational awareness system. However, due to the added uncertainty of 3 m in the obstacle track introduced by the situational awareness system, the safety margin of the motion reference system needs to be increased from 11 m to 14 m to refine the autonomy system contract. The guarantees for the motion reference to coincide with the pre-planned path and the constraints on the speed reference are copied from the autonomy system component in order to refine its contract.

```
moplan_assumptions = True

moplan_guarantees =
    And(distance_2d(motion_ref.position,
        track.position) >= 14.0,
        distance_2d(motion_ref.position,
        path) == 0,
        motion_ref.speed >= -1.0,
        motion_ref.speed <= 3.0)
```

Step 3: Define test setups for simulation-based testing

Next, we proceed to Step 3 of the methodology, where the test setups for simulation-based contract compliance checking are defined. For the milliAmpere II case, there already exists an extensive set of simulators used for development and verification. The simulators are developed in a collaboration between NTNU, the autonomous ferry start-up Zeabuz and the classification society DNV in the TRUSST research project (RCN, 2021). We will use these simulators as a basis for this section, and show how they can be configured to define test setups for contract compliance checking.

An overview of the full simulator setup for milliAmpere II is given in Fig. 9. The simulator is implemented over several different platforms. The traditional maritime simulation models are implemented on the Open Simulation Platform (OSP) (Smogeli et al., 2020), which is an open standard for co-simulation of maritime systems built on the FMI standard (Blockwitz et al., 2012). The 3D rendering of the virtual world and sensor models for the exteroceptive sensors are simulated on Gemini, which is an open-source simulation platform for autonomous vessels (Vasstein et al., 2020). Gemini is built on the Unity game engine, and its exteroceptive sensor models are based on a novel methodology for simulating electromagnetic radiation sensors using game engine technology (Vasstein et al., 2020). The autonomy software runs on the ROS, and the simulator supports SiL simulation using exact replicas of the autonomy software. This is also true for the COTS motion control system, however, this runs on the proprietary platform of the vendor. Some snapshots from the simulator in action are given in Fig. 10.

Recall that the goal of the simulation-based testing of a given component is to verify that the component complies with its contract,

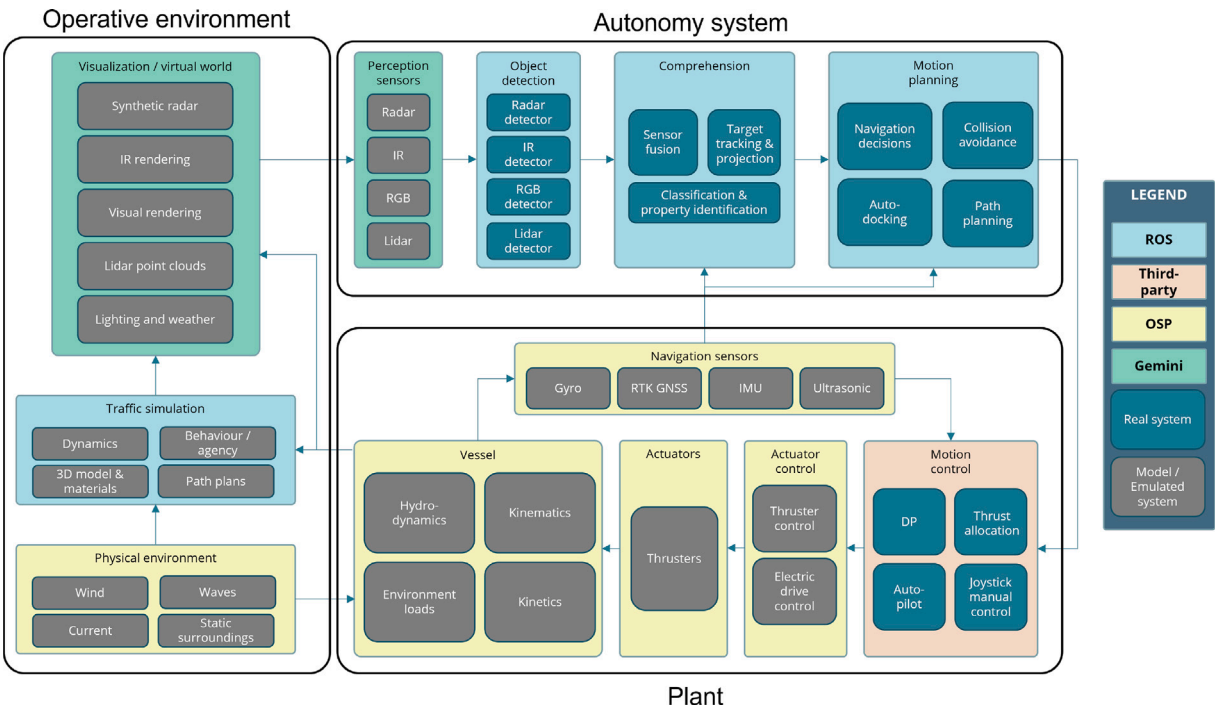

Fig. 9. Overview of the milliAmpere II simulator. The black boxes indicate which parts of the simulator correspond to the component structure we have defined in Fig. 8. In the autonomy system, the perception sensors, object detection, and comprehension modules together make up the situational awareness component.

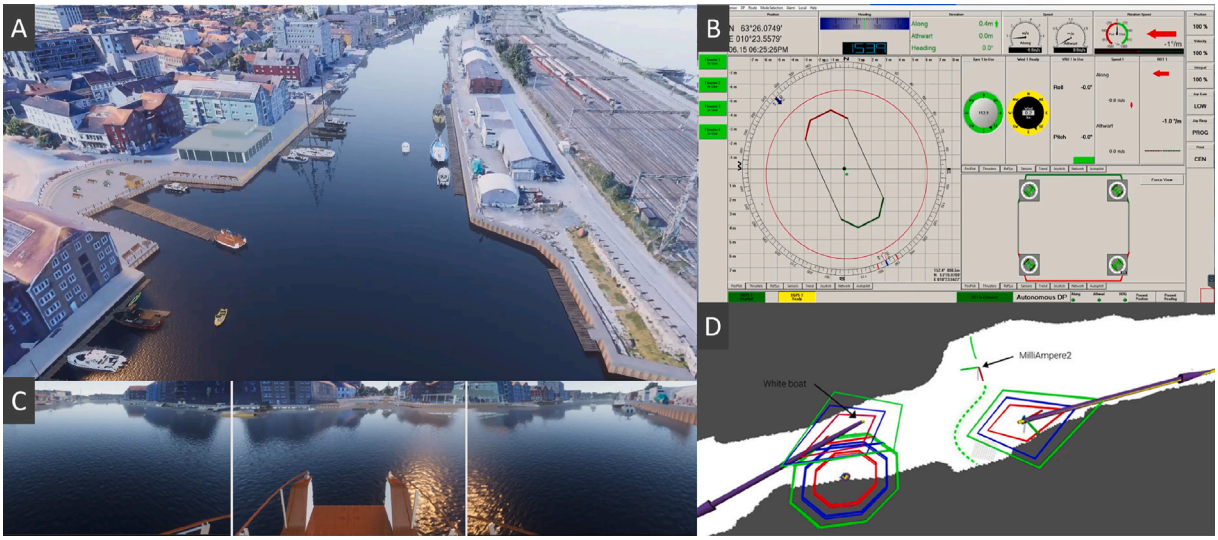

Fig. 10. Snapshots from the milliAmpere II simulator. (A) The 3D rendering of the operation area and the ferry. (B) The interface for the motion control system, including DP, thruster control, and navigation. (C) The output of the simulated RGB cameras. (D) A map of the situational awareness, where the detections from the simulated camera, lidar and radar sensors are fused and used to estimate the pose and speed of obstacles. Also shown in D are the safety margin zones around each obstacle.

as illustrated in Fig. 5. To achieve this, we need a simulation model of the component under test and a simulated test environment to generate the test signals on the in-ports of the component. Together, this defines the test setup for the component. We believe that it is advantageous to keep the test environment of the component as simple as possible in order to focus the testing to find inherent weaknesses in the component under test. This both simplifies the simulator and reduces the size of the test space for the component. Testing components in more realistic environments is of course also very important, however, this will be done when testing at the higher abstraction levels, where the component will be integrated with other components and thus be tested in a more realistic environment.

The full setup of Fig. 9 could represent the test setup for the top-level component since it simulates all parts of the component and its operative environment. To define the test setup for components at lower abstraction levels, we can use this setup as a basis, remove all modules which are deemed irrelevant and simplify all connected modules as much as possible. Fig. 11 shows the test setup for contract compliance checking of the autonomy system component. The vessel motion of the ferry is set equal to the motion reference output of the motion planning component, thereby we are assuming perfect trajectory tracking of the plant. The main objective of this setup is to test the integrated motion planning and situational awareness system for collision avoidance. Fig. 12 shows the test setup for contract

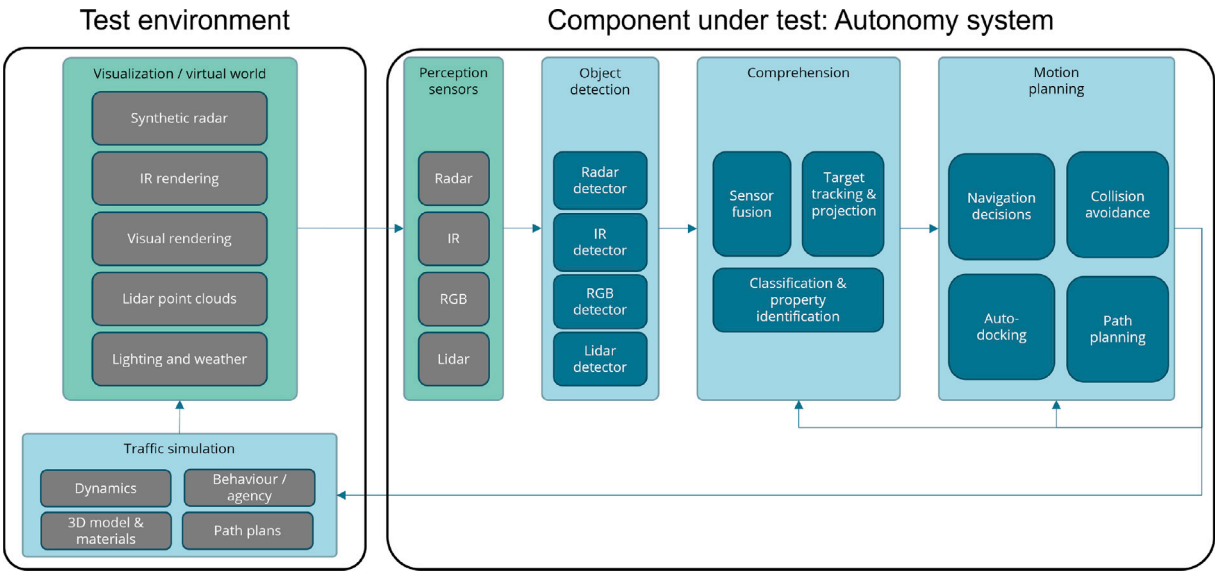

Fig. 11. Test setup for contract compliance checking of the autonomy system component.

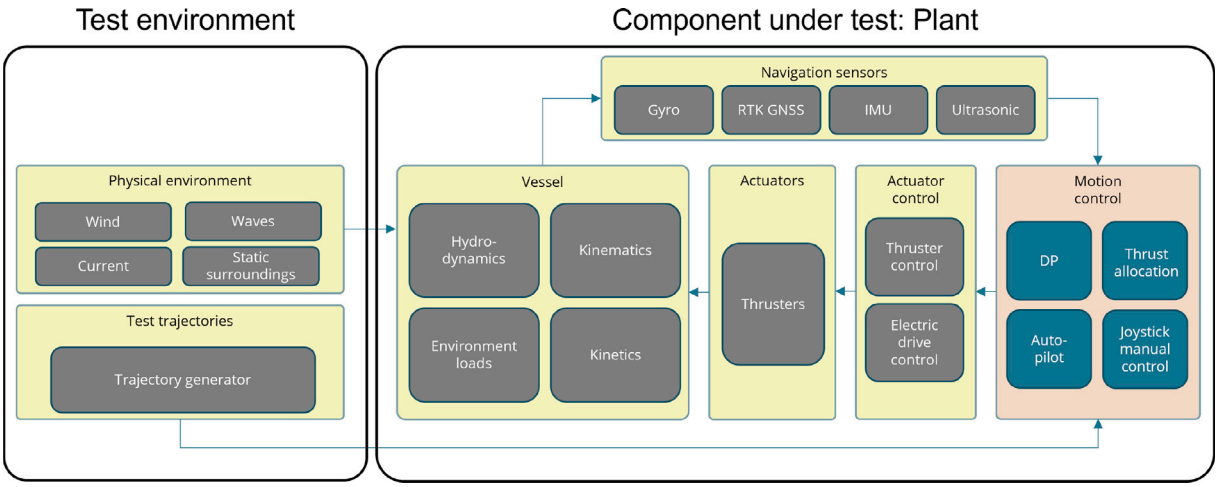

Fig. 12. Test setup for contract compliance checking of the plant component.

compliance checking of the plant component. The main objective of this setup is to test the trajectory tracking capabilities of the plant under different environmental loads and reference trajectories. To this end, we have defined a *test trajectories* module which generates motion reference inputs to the plant. Fig. 13 shows the test setup for contract compliance checking of the motion planning component. In this setup we are simplifying the test environment to use perfect tracks and perfect trajectory tracking, thus reducing both the situational awareness system and the plant to identity/pass through. The main objective of this setup is to verify collision avoidance for the motion planning component. Being very lightweight, this setup offers fast simulations and corresponding high test coverage. Finally, the test setup for the situational awareness component is shown in Fig. 14. The main objective of this setup is to verify the obstacle tracking capabilities of the situational awareness component. We are using the *test trajectories* module to generate the ferry motion.

Step 4: Recursive contract-based verification

With the component structure, contracts, and test setup in place, we can readily apply the recursive contract-based verification procedure of Algorithm 1 to the milliAmpere II system to achieve contract-based system verification. To demonstrate contract-based verification, we will in this section go through the step-by-step computations of Algorithm 1 applied to the components and contracts developed in the case study.

The entry point for the verification is applying *verifyComponent()* on the top-level milliAmpere II component. The first step is contract compliance checking. This involves running simulations with the test setup of Fig. 9 to verify that all behaviors σ satisfy $\varphi_A(\sigma) \rightarrow \varphi_G(\sigma)$. This amounts to running scenarios where the obstacle motion and environmental loads are within the assumptions and verify that the ferry keeps a 10 m distance to the obstacle and tracks the path with 1 m precision. After the contract compliance check is completed, the procedure continues into the branching statement of Algorithm 1, since the top-level component is composite. First, the composite contract of the sub-components is constructed by composing the contracts of the autonomy system and the plant. Then, the procedure performs the refinement check by passing the theorems $\varphi_A \rightarrow \varphi_A^{comp}$ and $\varphi_G^{comp} \rightarrow \varphi_G$ to Z3's *prove()* function. This yields a *proved* result and thereby verifies that the composite contract of the autonomy system and plant refine the top-level contract and that the contracts of the autonomy system and plant are compatible.

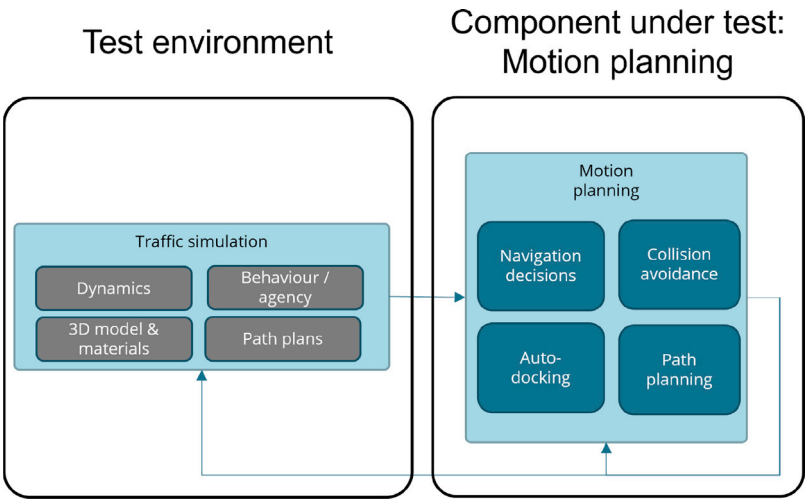

Fig. 13. Test setup for contract compliance checking of the motion planning component.

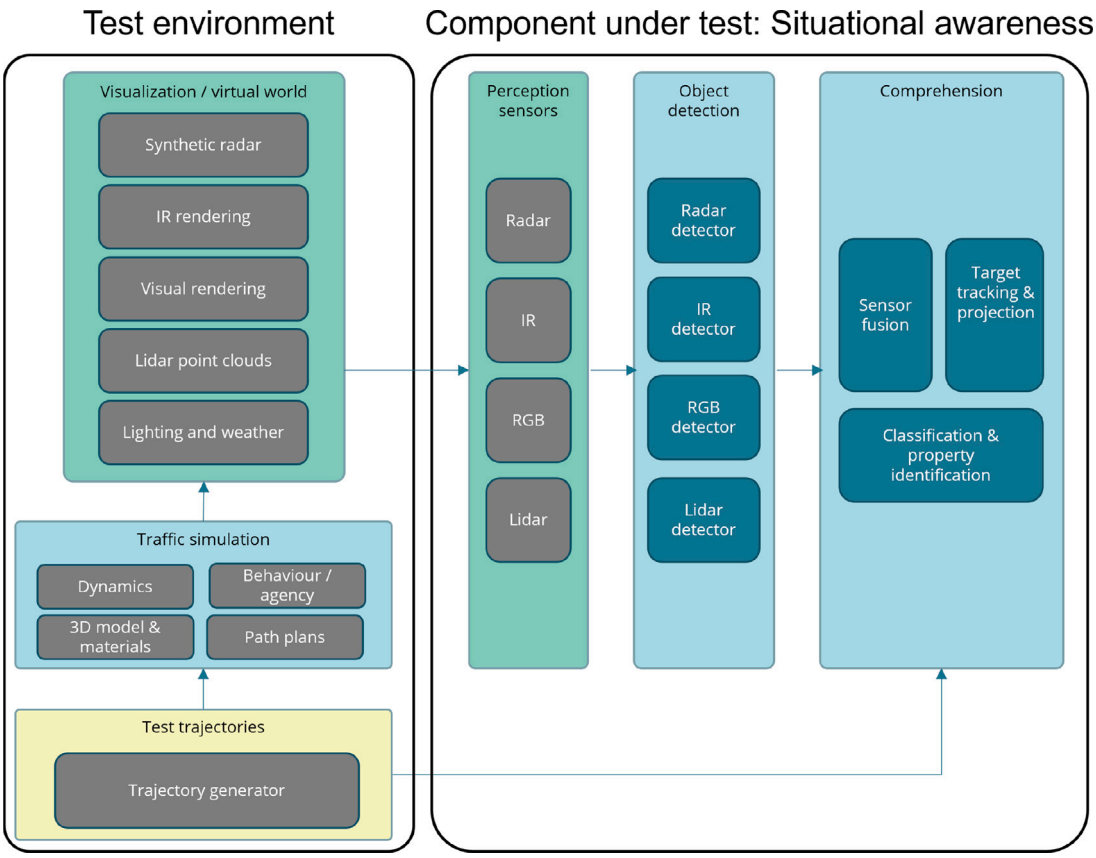

Fig. 14. Test setup for contract compliance checking of the situational awareness component.

The next step in Algorithm 1 is to recursively apply *verifyComponent()* to the sub-components of the top-level component. The procedure begins by applying it to the plant component. First, the contract compliance check is performed, this time using the test setup of Fig. 12. After completing the contract compliance checking the recursion breaks, since the plant is an atomic component. The procedure then moves on to applying *verifyComponent()* to the autonomy system component. This begins with contract compliance checking using the test setup of Fig. 11. Since the autonomy system component is composite, the procedure continues into the branching statement of Algorithm 1, which composes the contracts for the situational awareness and motion planning components and checks that the composite contract refines the autonomy system contract. Passing refinement checking theorems to Z3's *prove()* function again yields a *proved* result.

Finally, we recursively apply *verifyComponent()* to the situational awareness and motion planning components. The contract compliance checks are executed using the test setups of Figs. 14 and 13, respectively. Since both these components are atomic, the recursion breaks and Algorithm 1 terminates. The entire formal reasoning for the case study, including definition, saturation, and composition of contracts in addition to the refinement proofs, was executed in 4.9 s on a laptop with an Intel Core i9-9980HK CPU.

5. Discussion

The results from the case study indicate that the proposed contract framework and verification methodology overall achieve the goal of modular verification. Next, we discuss some challenges we encountered and some possible solutions, as well as directions for future work.

We experienced the Z3 first-order logic language as a natural and expressive syntax for specifying contracts. However, we encountered some difficulty in the automatic theorem proving of the refinement checks. When calling Z3's *prove()* function, a successful run returns either *proved* or *could not prove* with a counterexample. However, in some cases, it returns *unknown* or runs seemingly indefinitely without returning a result. In these cases, it can be hard and time-consuming to debug. It is well known that Z3 can be sensitive to seemingly unimportant changes in the theorem formulation, such as the order in which arguments are given. Some trial and error can therefore be necessary to get a successful run. For proofs including boolean variables and linear constraints over real variables, Z3 appeared to be very capable and scalable. The proofs involving nonlinear functions over real-valued variables did, however, also pose some difficulty. An example of this is the proofs involving the *distance_2d()* function, which is nonlinear due to the quadratic terms and the square root operation. It was not unexpected that this was problematic, as automatic theorem proving of such problems is a very hard computational problem that is undecidable in general. To get successful proofs here, we sometimes had to manually split the proofs into smaller and more manageable pieces which were proved independently. Z3 offers functionality for the user to input proof tactics, which may increase the capabilities to prove hard problems. An alternative approach to enhance the capabilities to prove nonlinear real-valued theorems is to use an interactive theorem prover, such as Isabelle (Paulson, 1994) or Coq (Bertot, 2008). These are generally capable of proving more complex problems, but also require more expertise and input from the user.

In this work, a critical success factor for a contract-based verification methodology is that it is scalable enough to give value to complex industrial autonomy use cases. In order to present a case study in full in this paper, the case study had to be simplified in several aspects. The components were simplified versions of the real components in the milliAmpere II system and not all interactions between components were considered. The operative environment model was also simplified. The contracts only considered certain aspects of the verification scope, and important aspects such as passenger comfort, reliability, fault handling, and timing were not considered. The refinement of the components was also stopped at a quite high level of abstraction. Because of these simplifications, the scalability of the methodology to industrial use cases cannot be concluded based on the case study only. Our intuition is, however, that such a structured approach is necessary to manage the complexity of autonomous vessel systems, and that scalability is, to a large degree, dependent on having sufficient tool support for defining and managing a large number of components and contracts in an enterprise setting. For the time being, the biggest obstacle to scalability seems to be the formal compositional reasoning, as discussed above. We do, however, believe that having a structured framework for design and verification using the concepts of assume-guarantee contracts, refinement, and composition, can provide great value to the design and verification process, even if the contracts are expressed in natural language and the compositional reasoning is informal.

Another motivation for doing modular verification is that it enables more efficient change management. Since autonomous vessels are highly software intensive they will likely be subject to frequent software updates. After a software update, the system will need to be re-verified. Re-verifying the entire system is a huge task, and doing this for every update is not desirable. The recursive contract-based procedure of Algorithm 1 can be extended to support selective re-verification, that is, only re-verify the parts of the system which are affected by the change. A straightforward approach to this is to add a hash to each component. The hash should be computed based on the component implementation, port definition, parameters, and contract. For composite components, the hash should also be computed based on the hash of all sub-components. The *verifyComponent(Component M)* procedure can then check if the hash of the M has changed since the last verification and terminate immediately if it has not changed. Suppose for instance a software update has been made to the situational awareness component in Fig. 8. This would change the hash of the situational awareness, autonomy system, and milliAmpere II components, but not the motion planning and plant components. The re-verification would thus re-verify the situational awareness component, its integration into the autonomy system, and its integration into the milliAmpere II component, but would skip the re-verification of the motion planning and plant components. If the contract compliance checking for the changed component is exhaustive, it would be sufficient to only re-verify that component. However, this will in general not be the case when using simulation-based contract compliance checking.

As introduced in Section 1, a key challenge for verifying autonomous systems is that they extensively sense and interact with the open environment, and there will be an infinite number of scenarios that are unknown during design. Such scenarios are often referred to as *edge cases*. This challenge is well-known from automotive autonomy, which has approached the challenge by extensive use of machine learning, with the hope that vast amounts of training data will expose the self-driving system to a sufficient number of edge cases. This approach has not yet been successful (Koopman, 2021). We believe that contracts may offer an alternative solution. While it is very hard to design an autonomous system that will act correctly in all edge cases, it may be feasible to design a system that can detect when a situation is outside the ODD and do something safe, such as going to a minimum risk condition (MRC). Compared to the automotive application, the safety margins generally are larger, and the speed is generally lower in maritime applications. Therefore, it may possible to ensure safety by going to an MRC. Online monitoring of the assumptions of the top-level contract, which specify the ODD, can implement such a system. We therefore consider online monitoring of assumptions and the use of this in the high-level decision making as an important topic for future work.

Finally, there are some directions in which the methodology can be extended. As mentioned in Section 2.1, we only consider synchronous behaviors. The extension to asynchronous behavior is a natural next step. Extensions to more expressive contracts is also a possible direction for future work. Examples include temporal contracts, using e.g. a temporal logic, and the use of quantifiers in the first-order logic formulas. Using the concept of different *viewpoints*, as proposed by Nuzzo et al. (2015), to combine different orthogonal aspects in the contracts is another interesting extension. A critical aspect of the proposed methodology is that it is dependent on completeness in the top-level contracts to achieve the verification objective. Furthermore, generating sufficiently many relevant test scenarios for the simulation may be demanding. As we briefly mentioned in Section 3, risk analysis may provide an important basis for defining safety requirements for the system, as well as for developing and selecting the test scenarios. We see this as an important direction for future work.

6. Conclusions and further work

This paper has introduced contract-based methods to the design and verification of autonomous vessels. We have presented a framework for specifying components and contracts based on the automatic theorem prover Z3 and a contract-based verification methodology using this framework. The contract-based verification methodology approaches compositional reasoning by constructing a composite assume-guarantee contract. Using the Z3 automated theorem prover, formal and automatic refinement checking has been achieved. Furthermore, we have shown how simulation-based testing can be used in conjunction with a

contract-based framework in a mutually beneficial way. The contract-based verification method was ultimately stated as a concise recursive procedure for system verification. The contract framework and verification procedure were demonstrated in a case study with the autonomous passenger ferry milliAmpere II.

The discussion highlighted the need for further work in the automatic theorem proving of the compositional reasoning. In particular, methods to better handle proofs including nonlinear functions over real variables should be investigated. Use of Z3's proof tactics or interactive theorem provers were suggested as possible directions for future work. The discussion also suggested possible extensions to the methodology, including online monitoring of assumptions and investigating the use of risk analysis methods as a basis for defining the top-level contract and test scenarios. Finally, the discussion highlights the need for demonstrating the use of the contract-based framework in an industrial-scale project to investigate the scalability of the method. Our hope is that our introduction of contract-based methods to maritime autonomy will trigger more research and development in this direction and ultimately contribute to safer and more robust autonomous vessel control systems.

CRediT authorship contribution statement

Tobias Rye Torben: Conceptualization, Methodology, Formal analysis, Writing – original draft. **Øyvind Smogeli:** Conceptualization, Writing – review & editing. **Jon Arne Glomsrud:** Conceptualization, Writing – review & editing. **Ingrid B. Utne:** Conceptualization, Writing – review & editing, Funding acquisition. **Asgeir J. Sørensen:** Conceptualization, Writing – review & editing, Funding acquisition.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

No data was used for the research described in the article.

Acknowledgments

The work by T.R. Torben, I.B Utne, and A.J. Sørensen is partly sponsored by the Research Council of Norway through the Centre of Excellence funding scheme, project number 223254, AMOS, Norway, and project ORCAS, Norway with project number 280655. The work by Øyvind Smogeli and Jon Arne Glomsrud is sponsored by the Research Council of Norway through the TRUSST project with project number 313921.

References

Abrial, J.R., 2011. Modeling in event-b: System and software engineering. Cambridge University Press.

Bakdi, A., Glad, I.K., Vanem, E., 2021. Testbed scenario design exploiting traffic big data for autonomous ship trials under multiple conflicts with collision/grounding risks and spatio-temporal dependencies. IEEE Trans. Intell. Transp. Syst. (June 2019), 1–17. http://dx.doi.org/10.1109/TITS.2021.3095547.

Benveniste, A., Caillaud, B.t., Ferrari, A., Mangeruca, L., Passerone, R., Sofronis, C., 2008. Multiple viewpoint contract-based specification and design. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (Eds.), Formal Methods for Components and Objects. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 200–225.

Benveniste, A., Caillaud, B.t., Nickovic, D., Passerone, R., Raclet, J.-B., Reinkemeier, P., Sangiovanni-Vincentelli, A., Damm, W., Henzinger, T.A., Larsen, K.G., 2018. Contracts for system design. volume 12. Now.

Bertot, Y., 2008. A short presentation of coq. In: Mohamed, O.A., Muñoz, C., Tahar, S. (Eds.), Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), Vol. 5170 LNCS. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 12–16. http://dx.doi.org/10.1007/978-3-540-71067-7{_}3.

Blockwitz, T., Otter, M., Akesson, J., Arnold, M., Clauss, C., Elmqvist, H., Friedrich, M., Junghanns, A., Mauss, J., Neumerkel, D., Olsson, H., Viel, A., 2012. Functional mockup interface 2.0: The standard for tool independent exchange of simulation models. In: Proceedings of the 9th International MODELICA Conference, September 3-5, 2012, Munich, Germany, Vol. 76. pp. 173–184. http://dx.doi.org/10.3384/ecp12076173.

Brekke, E.F., Eide, E., Eriksen, B.-O.H., Wilthil, E.F., Breivik, M., Skjellaug, E., Helgesen, O.K., Lekkas, A., Martinsen, A.B., Thyri, E.H., Torben, T., Veitch, E., Alsos, O.A., Johansen, A., 2022. Milliampere: An autonomous ferry prototype. J. Phys. Conf. Ser. 2311 (1), 012029.

Chaal, M., Valdez Banda, O.A., Glomsrud, J.A., Basnet, S., Hirdaris, S., Kujala, P., 2020. A framework to model the STPA hierarchical control structure of an autonomous ship. Saf. Sci. 132 (July), 104939. http://dx.doi.org/10.1016/j.ssci.2020.104939.

Cimatti, A., Tonetta, S., 2012. A property-based proof system for contract-based design. In: Proceedings - 38th EUROMICRO Conference on Software Engineering and Advanced Applications, SEAA 2012. IEEE, pp. 21–28. http://dx.doi.org/10.1109/SEAA.2012.68.

Clarke, E.M., 1997. Model checking. In: Ramesh, S., Sivakumar, G. (Eds.), Foundations of Software Technology and Theoretical Computer Science. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 54–56.

Clarke, E.M., Long, D.E., McMillan, K.L., 1989. Compositional model checking. In: Proceedings. Fourth Annual Symposium on Logic in Computer Science. pp. 353–362. http://dx.doi.org/10.1109/LICS.1989.39190.

Foster, S., Gleirscher, M., Calinescu, R., 2020. Towards deductive verification of control algorithms for autonomous marine vehicles. In: Proceedings of the IEEE International Conference on Engineering of Complex Computer Systems. pp. 113–118.

Hake, G., Hohl, C.P., Hahn, A., 2021. Continuous contract based verification of updates in maritime shipboard equipment. http://dx.doi.org/10.3390/jmse9070688, URL: https://doi.org/10.3390/jmse9070688.

Kahn, G., 1974. The Semantics of a Simple Language for Parallel Programming. Technical Report.

Kapinski, J., Deshmukh, J.V., Jin, X., Ito, H., Butts, K., 2016. Simulation-based approaches for verification of embedded systems. IEEE Control Syst. Mag. 36 (November).

Koopman, P., 2021. SOTIF & edge cases. URL: https://users.ece.cmu.edu/~koopman/lectures/ece642/L103_SOTIF_EdgeCases.pdf.

Maler, O., Nickovic, D., 2004. Monitoring temporal properties of continuous signals. In: Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems. Springer, pp. 152–166.

Meyer, B., 1992. Applying 'design by contract'. Computer 25 (10), 40–51.

de Moura, L., Bjørner, N., 2008. Z3: An efficient SMT solver. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 337–340.

Murray, B., Rødseth, O.J., Nordahl, H., Wennersberg, L.A.L., Pobitzer, A., Foss, H., 2022. Approvable AI for autonomous ships: Challenges and possible solutions. In: Proceedings of the 32nd European Safety and Reliability Conference (ESREL 2022). pp. 1975–1982. http://dx.doi.org/10.3850/978-981-18-5183-4.

NMD, 1990. Nordisk Båtstandard for Yrkesbåter Under 15 Meter. Technical Report.

Nuzzo, P., Sangiovanni-Vincentelli, A.L., Bresolin, D., Geretti, L., Villa, T., 2015. A platform-based design methodology with contracts and related tools for the design of cyber-physical systems. Proc. IEEE 103 (11), 2104–2132. http://dx.doi.org/10.1109/JPROC.2015.2453253.

Nuzzo, P., Xu, H., Ozay, N., Finn, J.B., Sangiovanni-Vincentelli, A.L., Murray, R.M., Donzé, A., Seshia, S.A., 2014. A contract-based methodology for aircraft electric power system design. IEEE Access 2, 1–25. http://dx.doi.org/10.1109/ACCESS.2013.2295764.

Paulson, L.C., 1994. Isabelle: A Generic Theorem Prover, Vol. 828. Springer Science & Business Media.

Pedersen, T.A., Glomsrud, J.A., Ruud, E.L., Simonsen, A., Sandrib, J., Eriksen, B.O.H., 2020. Towards simulation-based verification of autonomous navigation systems. Saf. Sci. 129 (December 2019), 104799. http://dx.doi.org/10.1016/j.ssci.2020.104799.

RCN, 2021. TRUSST: Assuring trustworthy, safe and sustainable transport for all. URL: https://prosjektbanken.forskningsradet.no/project/FORISS/313921?Kilde=FORISS&distribution=Ar&chart=bar&calcType=funding&Sprak=no&sortBy=date&sortOrder=desc&resultCount=30&offset=210&TemaEmne.2=N/T1/ring+og+handel.

Rokseth, B., Haugen, O.I., Utne, I.B., 2019. Safety verification for autonomous ships. In: MATEC Web of Conferences, Vol. 273. p. 02002. http://dx.doi.org/10.1051/matecconf/201927302002.

Rokseth, B., Utne, I.B., 2019. Deriving safety requirement hierarchies for families of maritime systems. Trans. R. Inst. Nav. Archit. A 161, A229–A243. http://dx.doi.org/10.3940/rina.ijme.2019.a3.526.

Sangiovanni-Vincentelli, A., Damm, W., Passerone, R., 2012. Taming Dr. Frankenstein: Contract-based design for cyber-physical systems. Eur. J. Control 18 (3), 217–238. http://dx.doi.org/10.3166/EJC.18.217-238, URL: http://dx.doi.org/10.3166/ejc.18.217-238.

Shokri-Manninen, F., Vain, J., Waldén, M., 2020. Formal verification of COLREG-based navigation of maritime autonomous systems. In: Lecture Notes in Computer Science. pp. 41–59.

Smogeli, O., 2015. Managing DP system software - A life-cycle perspective. IFAC-PapersOnLine 48 (16), 324–334.

Smogeli, O., Ludvigsen, K.B., Jamt, L., Vik, B., Nordahl, H., Kyllingstad, L.T., Yum, K.K., Zhang, H., 2020. Open simulation platform – An open-source project for maritime system co-simulation. In: 19th Conference on Computer and IT Applications in the Maritime Industries. pp. 239–253.

Smogeli, O., Skogdalen, J.E., 2011. Third party HIL testing of safety critical control system software on ships and rigs. In: Offshore Technology Conference. One Petro, pp. 839–845.

Thyri, E.H., Breivik, M., Lekkas, A.M., 2020. A path-velocity decomposition approach to collision avoidance for autonomous passenger ferries in confined waters. IFAC-PapersOnLine 53 (2), 14628–14635. http://dx.doi.org/10.1016/j.ifacol.2020.12.1472, URL: https://doi.org/10.1016/j.ifacol.2020.12.1472.

Torben, T.R., Glomsrud, J.A., Pedersen, T.A., Utne, I.B., Sørensen, A.J., 2022a. Automatic simulation-based testing of autonomous ships using Gaussian processes and temporal logic. J. Risk Reliab. 1–21.

Torben, T.R., Smogeli, O., Utne, I.B., Sørensen, A.J., 2022b. On formal methods for design and verification of maritime autonomous surface ships. In: Proceedings of the 7th World Maritime Technology Conference. Copenhagen, pp. 251–262.

Utne, I.B., Rokseth, B., Sørensen, A.J., Vinnem, J.E., 2020. Towards supervisory risk control of autonomous ships. Reliab. Eng. Syst. Saf. 196 (June 2018), 106757. http://dx.doi.org/10.1016/j.ress.2019.106757, URL: https://doi.org/10.1016/j.ress.2019.106757.

Vasstein, K., 2021. A high fidelity digital twin framework for testing exteroceptive perception of autonomous vessels.

Vasstein, K., Brekke, E.F., Mester, R., Eide, E., 2020. Autoferry gemini: A real-time simulation platform for electromagnetic radiation sensors on autonomous ships. IOP Conf. Ser.: Mater. Sci. Eng. 929 (1), http://dx.doi.org/10.1088/1757-899X/929/1/012032.

Woerner, K., Benjamin, M.R., Novitzky, M., Leonard, J.J., 2019. Quantifying protocol evaluation for autonomous collision avoidance: Toward establishing COLREGS compliance metrics. Auton. Robots 43 (4), 967–991. http://dx.doi.org/10.1007/s10514-018-9765-y.