Component-based Design and Verification in X-MAN

Nannan He, Daniel Kroening, Thomas Wahl, Kung-Kiu Lau, Faris Taweel, Cuong Tran, Philipp Rümmer, Sanjiv S Sharma

Component-based Design and Verification in X-MAN

Nannan He, Daniel Kroening, Thomas Wahl
Oxford University

Kung-Kiu Lau, Faris Taweel, Cuong M. Tran
University of Manchester

Philipp Rümmer
Uppsala University

Sanjiv S. Sharma
Airbus Operations Limited

***Abstract*—Compositionality has the potential to enable a step-change in the scalability of formal design and verification methods for industrial-scale systems: by designing systems in a compositional manner, components can be modelled, specified, implemented, and verified independently and in parallel by different teams, leading to significant gains in terms of productivity and the ability to reuse components. We discuss why component-based frameworks have, up to now fallen short of meeting those expectations, and present the X-MAN framework; a component-based framework and development methodology that has been designed to overcome limitations of previous solutions. Walking through an industrial case study, we illustrate architecture, specification, detailed design, and implementation of systems in X-MAN. Correctness and reliability concerns are addressed uniformly within X-MAN through integration with existing static analysis tools for functional and extra-functional properties.**

***Keywords*-Component-based modeling; compositional verification and validation; formal methods.**

I. Introduction

An increasing number of computer systems are embedded into other devices, including in numerous safety-critical domains. At the same time, the complexity of the software in these systems is growing exponentially. One approach to manage this complexity in the design phase is to apply a component-based software development approach, which promises time-savings, cost-savings, and increased productivity via component reuse [21]. However, using off-the-shelf components properly and safely is a serious challenge, which implies the significant need to specify and verify the components and their assemblies, up to the level of the full system (built from components) to assure functional and non-functional correctness [10].

As an algorithmic formal verification method, *Model Checking* [9] automatically assesses whether a finite-state model of a system under test satisfies a given formal specification. Significant advances have been achieved in the development of model checking. However, scalability is still one of the main obstacles in the application of Model Checking to verify complex industrial systems. One key approach in achieving scalability is "divide-and-conquer", which typically involves two basic steps. Firstly, the system is decomposed into a set of smaller components, in order to apply Model Checking to individual components (of managable size) in isolation. Secondly, the system-level property is established by assembling the verification results for its components without analyzing the whole system.

The research leading to these results has received funding from the ARTEMIS Joint Undertaking under grant agreement no. 100016 and from the Technology Strategy Board, UK. It has been carried out as part of the the CESAR project (http://www.cesarproject.eu/).

The ARTEMIS CESAR project aims to improve the cost-efficiency of processes and methods for safety-critical embedded systems development by an order of magnitude. One objective is to bring significant innovations in some of the systems engineering disciplines with most scope for improvement such as component-based engineering. Thus, component-based development, which facilitates system exploration and provides means for incremental verification, validation and certification, is one of the main focuses in CESAR. We present a component-based design and verification methodology based on X-MAN arising from and supported by the CESAR project, and give a case study of its application in a real-world embedded system from industry.

X-MAN is a framework for compositional software design developped at the University of Manchester [17]. Its features include, among others, (i) strong separation of components, that is, components are strictly separated with respect to data and side-effects of execution at the functional level, and (ii) that coordination control among a set of components is entirely governed by *composition connectors*. These features suggest that the X-MAN framework is well-suited to compositional verification. Our aim is to develop a new compositional design and verification methodology based on X-MAN using automated formal techniques, and to investigate its applicability to safety-critical embedded systems, specifically in avionics and automotive control.

Related Work

X-MAN is not the only framework of this kind. The *Component Formal Reasoning Technology* (ComFoRT) [7] is a modelling and reasoning framework developed at the SEI with the aim of predictability in the construction of component-based systems, that is, predicting the behavior of a component-based system prior to implementation based on properties of the components. The strategy is to formalize a particular component-based idiom [22] in a custom component language for modeling design specifications, and then to apply model checking. Although ComFoRT is comprehen-

sive, its actual applicability to embedded systems has not been studied extensively. In [19], Quinton et al. proposed a contract-based reasoning framework for the component-based design of complex, hierachically defined systems. It is based on an existing component framework for constructing systems called BIP (Behavior, Interaction, Priority) [2]. The composite connectors in BIP have the formally defined semantics [5]. This framework supports rigorous component-based design [1] and verification [4] in various applications, such as the componentization [3] of existing functional modules [12] in apply to robotic systems. In comparison, while the compositional verification methodology proposed in this paper is mainly based on the X-MAN component-based framework, it can be generalized easily to other similar component-based frameworks.

With the active development of component technologies, formal verification for component-based designs is gaining prominence and modular reasoning is key to its scalability [11]. One of the issues is the effort required to specify the individual components, which has stimulated research on automatically generating modular assumptions under which a given component satisfies a desired property, enabling automated computation of contracts for individual components. These techniques are out of the scope of this paper, and we only mention [13] as an exemplar.

Outline

This paper is organized as follows: in Section 2, we first present the X-MAN component design methodology. Next, in Section 3, we elaborate on the formal semantics for X-MAN, which are the foundation of compositional formal verification described in Section 4. We then present the case study extracted from a real-world application in avionics, apply the X-MAN modeling framework and provide results of the application of the compositional verification methodology to the case study in Section 5. Finally in the Appendix, we show some screenshots of the X-MAN tool when applied to the case study.

II. The X-MAN Component Design Methodology

In the X-MAN component-based approach, components are constructed from two kinds of basic entities: (i) *computation units*, and (ii) *connectors*. A computation unit *U* encapsulates computation. It provides a set of methods. *Encapsulation* means that *U*'s methods do not call methods in other computation units; rather, when invoked, all their computation occurs inside *U*. Thus, *U* can be thought of as a class that does not call methods in other classes.

There are two kinds of connectors: (i) *invocation*, and (ii) *composition*. An invocation connector is connected to a computation unit *U* so as to provide access to the methods of *U*.

A composition connector encapsulates *control*. It is used to define and coordinate the control for a set of components. For sequencing, we use the *pipe* and *sequencer* connectors, and for branching, we use the *selector* connector. A *pipe* connector that composes components $C_1, ..., C_n$ can call methods in $C_1, ..., C_n$ in that order, and pass the results of calls to methods in C_i to those in C_{i+1}. A *sequencer* connector is the same as a *pipe* but does not pass the results of C_i to C_{i+1}. A *selector* connector that composes components $C_1, ..., C_n$ simply selects one component according to a selection condition.

Components are defined in terms of computation units and connectors. There are two kinds of components: (i) *atomic*, and (ii) *composite*. An atomic component consists of a computation unit with an invocation connector that provides an interface to the component. An atomic component is depicted in Fig. 1(a). A composite component consists of a set of components (atomic or composite) composed by a composition connector. The composition connector provides an interface to the composite. A composition connector is presented in Fig. 1(b) and a composite component composed from two atomic components and a connector is depicted in Fig. 1(c).

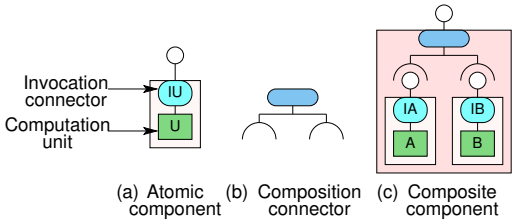


Figure 1. X-MAN component model.

Invocation and composition connectors form a hierarchy [16], i.e., composition is performed in a hierarchical manner. Furthermore, each composition preserves encapsulation. This kind of compositionality is the distinguishing feature of the X-MAN approach. An atomic component encapsulates computation, namely the computation encapsulated by its computation unit. A composite component encapsulates *computation* and *control*. The computation it encapsulates is that encapsulated in its sub-components; the control it encapsulates is that encapsulated by its composition connector. In a composite, the encapsulation in the sub-components is preserved. Indeed, the hierarchical nature of the connectors means that composite components are *self-similar* to their sub-components, i.e., composites have the same structure as their sub-components; this property provides a basis for hierarchical composition.

In general, a system constructed using this approach consists of a hierarchy of composition connectors sitting atop a flat layer of decoupled atomic components as shown in Fig. 3. The hierarchy of composition connectors totally encapsulates the control in the system, whilst the atomic components encapsulate the computation performed by the system.

III. Formal Semantics of X-MAN

A. The Meaning of a Component

Our modeling and verification methodology is agnostic to the programming language used to implement atomic components, the precise format of contracts used to specify components or sub-systems, and the language used to specify system properties to be verified. To ensure the overall soundness of our approach, we therefore designed a unifying, language-independent formal semantics for X-MAN. By mapping various design artifacts into this common formal framework, we are able to freely mix (at a conceptual level) different implementation languages, different specification languages, or hard- and software components within a single model.

In our semantics, a component, system, or sub-system T is a tuple $(S, S^0, Op, impl)$ consisting of

- a *state space* S, which is a non-empty (finite or infinite) set,
- a non-empty set $S^0 \subseteq S$ of *initial states*,
- a finite set Op of *operations* offered by the component,
- a mapping $impl$ from operations to *operation implementations*. The implementation $impl(op)$ of an operation $op \in Op$ is a relation between pre-states, inputs, post-states, and outputs

$$impl(op) \subseteq (\underbrace{S \times I_1 \times \cdots \times I_n}_{\text{Inputs}}) \times (\underbrace{S \times O_1 \times \cdots \times O_m}_{\text{Outputs}})$$

  i.e., is modelled to be potentially partial and potentially non-deterministic. If the context is clear, we also specify the signature of an operation as $op : I_1 \times \cdots \times I_n \to O_1 \times \cdots \times O_m$.

B. Construction of Components

Components can be constructed in a number different ways:

- *concrete* atomic components are implemented in programming languages such as C; the semantics and behavior of such a component is inherited from the semantics of C.
- *abstract* atomic components are defined declaratively, e.g., via a set of contracts consisting of pre- and post-conditions, similarly to the definition of abstract datatypes in terms of axioms. The semantics of an abstract component is derived from the semantics of the employed formal specification language (e.g., first-order logic).
- *composite* components are constructed by applying composition connectors, invocation connectors, or adapters to a set of simpler components. Conceptually, connectors and adapters are therefore mathematical functions operating on components as defined in Sect. III-A.

The formal semantics of connectors and adapters is unambiguously defined in terms of mathematical set theory. To generate executable code from X-MAN models, it is also necessary to provide concrete implementations (*realizations*) of the X-MAN connectors and adapters, which is done as part of the X-MAN modeling tool that is used for our experiments. Given an X-MAN model, and implementations of all concrete components in the C programming language, the tool is able to automatically generate a C implementation of the whole modeled system. It is possible to prove the the consistency of the connector realizations with their formal semantics, but this step is outside of the scope of this paper.

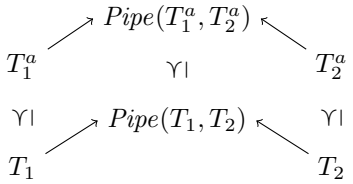
Examples of concrete and abstract components are given in Sect. V.

C. Component Refinement

The behavior of concrete and abstract components (e.g., a concrete implementation of a component in C, and an abstract description of the same component in terms of contracts) can be related using the notion of *refinement*. Intuitively, a concrete component T_1 *refines* an abstract component T_2 (written $T_1 \preceq T_2$) if T_1 has "less behavior" than T_2, i.e., all computation steps and outputs possible for T_1 are also possible for T_2, but not necessarily vice versa.

We use the notion of component refinement for various verification tasks: *vertical* verification means proving that a concrete component T (say, implemented in the C language) satisfies it's specification, i.e., does not violate any of the contracts that have been formulated for it. In terms of components, we say that the concrete component T *refines* the corresponding abstract component T^a defined by the contracts, i.e., $T \preceq T^a$.

Refinement also supports *horizontal* verification, which means to prove that the contracts of components entail desired properties of the system as a whole. Horizontal verification can be conducted by means of refinement, since we can observe that the X-MAN connectors and adapters are *monotonic* with respect to the refinement relation $\preceq$. For instance, assume that the relations $T_1 \preceq T_1^a$ and $T_2 \preceq T_2^a$ hold. This implies that the composite components obtained by applying the $Pipe$ connector are similarly related by $\preceq$, i.e., $Pipe(T_1, T_2) \preceq Pipe(T_1^a, T_2^a)$:



In other words, in order to prove that a system $Pipe(T_1, T_2)$ satisfies a certain safety property, it is sufficient to verify that the corresponding abstract system $Pipe(T_1^a, T_2^a)$ (obtained by composing the contracts that represent the components T_1^a and T_2^a) has this property.

To introduce refinement more formally, suppose two components $T_i = (S_i, S_i^0, Op, impl_i)$ (for $i \in \{1, 2\}$) with the same set Op of operations are given. We say that T_1 refines T_2, written $T_1 \preceq T_2$, if there is a relation $R \subseteq S_1 \times S_2$ such that (i) for every initial state $s_1 \in S_1^0$ there is an initial state $s_2 \in S_2^0$ such that $s_1 \, R \, s_2$; and (ii) for every operation $op \in Op$ the following property holds:

> whenever $impl_1(op)(s_1, \bar{i}, s_1', \bar{o})$ and $s_1 \, R \, s_2$ hold, there is a state $s_2' \in S_2$ such that $impl_2(op)(s_2, \bar{i}, s_2', \bar{o})$ and $s_1' \, R \, s_2'$.

In this case, we also say that T_1 refines T_2 with respect to the relation R. Note that $\preceq$ is a pre-order on components. Also note that our definition closely corresponds to the notion of *simulation relations* on transition systems.

In order to handle non-functional or extra-functional properties, a more general notion of refinement can be introduced (which is beyond the scope of this paper). In this generalisation, abstract operations are allowed to have a "richer" signature than the corresponding concrete operations. For instance, the parameters $preTime$ and $postTime$ described in Sect. IV are not explicitly present in concrete components; nevertheless, such parameters can be introduced explicitly at the level of abstract components to reason about timing. Other kinds of resources, such as program memory, can be handled in a similar way.

IV. Compositional Verification and Validation Methodology

A. Vertical and Horizontal Verification

Compositional verification requires an interface specification (defined in terms of a contract [18]) for individual components, in addition to the specification for the full system. With these two kinds of specifications, it is possible to establish that (i) individual components satisfy their contracts, and (ii) the system as a whole satisfies the system specification, based on the component contracts and the compositional relations of the components, using the assumption that these contracts have been verified in the first step. The compositional verification step does not rely on the concrete implementation of the components.

In X-MAN, the primary form of contracts is pre-/post-conditions and input/output paramenters of the component computation unit. We assume that these contracts are stored in a repository together with the component implementation, thus enabling reuse. Each component can have multiple contracts which specify different properties of the component. The pre-conditions of a contract specify the scenarios in which the contract applies; e.g., they could require that the input parameters of the operation (provided at runtime) are within legal ranges. Whenever the pre-conditions are satisfied, and the contract can be applied, the post-conditions of the contract specify the relationship between input and output parameters (computed by the operation), as well as the relationship between the pre- and the post-state of the component (in case of stateful components).

In addition to such *functional* properties, *timing* properties can also be specified in terms of the parameters *preTime* and *postTime* implicitly present for each component. The semantics of pre-/post-conditions is that of *partial correctness*, as no claim about termination of component operations is made.

Given an X-MAN model in which every atomic component is instrumented with a contract specification, compositional verification consists of two phases:

(i) *Vertical verification* ensures that the implementations of each atomic component satisfy their component contracts. To this end, the component implementation is instrumented with assertions generated from the component contracts. The assertions are subsequently verified using an automated software model checker. Such an instrumentation is possible both for functional contracts and specific non-functional contracts, such as those related to timing.

(ii) *Horizontal verification*: the atomic component contracts are used to derive or verify properties of the system. Such properties can also be described using pre-/post-conditions, but are given not for individual components but for the whole X-MAN model. The second phase, consequently, only relates the contracts specified at different levels of the system to each other. It is therefore sufficient to employ first-order decision procedures such as SMT solvers [15] (in contrast to the software model checkers required for the first phase). Depending on the composition connectors used in the system, it is also possible to compute the strongest contracts that hold for a given composition by means of quantifier elimination techniques.

Once both phases have been completed successfully, global correctness of the X-MAN model is guaranteed, in the sense that every concrete execution of the implementation satisfies the desired global system properties.

B. The X-MAN Verification Tools

Our verification methodology is implemented in the *X-MAN-Verifier* tool, developed by Oxford University and Uppsala University. The X-MAN-Verifier is able to verify both vertical and horizontal correctness properties of X-MAN models, with the help of different verification backends, and provides a common user interface to display the status of the various verification tasks, as well as diagnostic information in cases where verification fails.

Vertical verification: We use the bounded model-checker CBMC [8] for ANSI-C and C++ programs as backend for vertical verification. By exhaustively exploring the behaviour of programs up to a given loop bound, it can verify safety properties such as the adherence to array bounds (no buffer overflows), pointer safety, absence of exceptions, as

well as user-specified assertions or contracts. If a property does not hold, CBMC returns a counterexample, which is a trace that violates the specification. Counterexamples can often simplify debugging a faulty design or specification significantly.

Many embedded applications have strict real-time requirements. As a consequence, loop constructs in embedded software code often have a fixed bound on the number of iterations. CBMC is able to formally validate such bounds by means of *unwinding assertions*. Once the bound is established, it provides proof of the absence of errors. CBMC models machine integer and IEEE floating-point arithmetic accurately and can reason about machine-level artefacts such as bit-wise operators and integer overflows [6]. It is therefore capable of detecting a bugs that go unnoticed by many other formal verification tools.

Horizontal verification: The X-MAN-Verifier is also able to check refinement properties between contracts and system properties, thus implementing means of horizontal verification. Beyond the verification of system properties, the tool is able to automatically infer the *strongest properties* (strongest post-conditions) satisfied by the system operations, from the contracts specified for atomic system components.

To realize this functionality, the X-MAN-Verifier includes a complete set of formal models of the X-MAN connectors and adapters in terms of first-order logic. Reasoning about such formal models, as well as contracts and system properties, is done using the Princess theorem prover [20] for Presburger arithmetic and first-order logic.

V. A Case Study in Avionics

We conducted a case study on a representative avionics application – the Ground Fuel Transfer function of a large transport aircraft. It models the specific behaviours of the fuel management system when the aircraft is physically on the ground, as opposed to behaviours while the aircraft is in flight.

A. An overview of the component structure of the Ground Fuel Management System.

Recall that the behaviour of an X-MAN component is defined by means of a set of operations. The gound fuel transfer system consists of six selective top-level operations which are mutually execlusive (Fig. 2): *Automatic Refuel* (AR), *Manual Refuel* (MR), *Defuel* (DF), *Ground Transfer* (GT), *Shut-Off Test* (SOT) and *OFF*. Each of them is further composed of a set of sub-operations. In this paper, the sub-modules are shown for the *Manual Refuel* operation only, as shown in the largest box of Fig. 2, the *MR* operation is composed of four sub-operations: *Eval_Cond*, *Idle*, *In-Process*, and *Abort*. The last three sub-operations are also selectable on a mutually exclusive basis. *Eval_Cond* shall be called first to decide which of the rest three sub-operations is chosen to be executed next at the runtime. Furthermore, the *In-Process* consists of another four sub-components in sequence. They determine the respective operations of the central tank (*CT*), the left-wing tank (*LWT*), the right-wing tank (*RWT*), and the surge tank (*SP*) under the manual refuel operation mode. All tanks provide both fuel output via pumps and fuel inlet via valves, each being independently switchable by an external controller. This controller monitors the fuel flow between tanks, calculates the required tank-to-tank fuel transfers and sends the appropriate control signals to the pumps and valves of all tanks.

The internal compositional structure of the MR component is typical for the ground fuel management system. The other operations have a similar compositional architecture; for brevity, Fig. 2 contains the details of MR only. Moreover, the component-based implementation of MR exercises most features available in the X-MAN framework, making it a suitable exemplar to illustrate the proposed component-based design and verification approach.

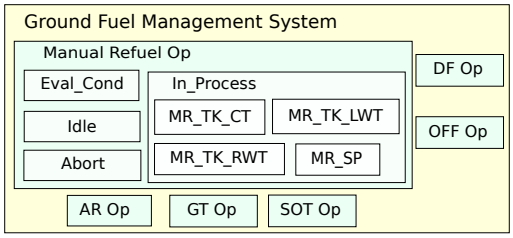


Figure 2. A functional composition overview of a ground fuel management system

In order to clearly describe the properties under verification, we first clarify the differences between two terms used in the system requirement specifications of our case study. 1). *State*: describes the (durable) condition after the system performs one or a sequence of operations following the entry condition. For instance, the *Inlet Valve* state of each tank shall be either *"SHUT"* or *"OPEN"*. 2). *Status*: It describes the extant condition of some physical component. In this case study, the status of each component is evaluated every cycle and given the value *NORMAL* or *FAILED*.

For verification purposes, most properties that we currently check are *safety* properties, which state that *"something bad must never happen"*, such as:

(i) When the system *input overflow_condition* is *True*, *MR* must be in the *Abort* operation mode no matter what other inputs are.
(ii) When the *Inlet Valve* state of a tank is *SHUT*, but the status of this *Valve* is *FAILED*, the fuel mass of this tank must never exceed a certain constant value C.
(iii) When the status of *MR* is *NORMAL*, the execution of *MR* must never be engaged for longer than 55 seconds.

These example properties above are at the different abstraction levels. (i) and (iii) refer to the entire system-level

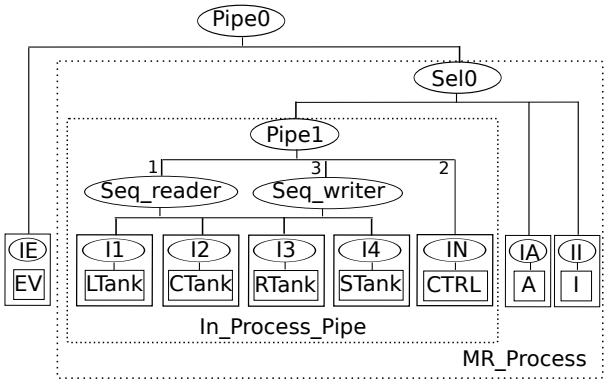


Figure 3. A component-based manual refuel system

properties of *MR*, while (ii) relates to the specific atomic operation of a tank which can be used directly to formulate the contract of an atomic component. Further discussions are given in the rest of the paper.

We implement a component-based version of the ground fuel management system by applying the X-MAN component framework, and verify the given properties of the system by using the compositional verification approach adapted to this framework. The details of the component implementation and compositional verification on the MR operation of this system are presented in the following subsections.

B. X-MAN Modeling of the Manual Refuel Example

In X-MAN component modelling, systems are built by composing component instances which are instantiated from component designs in the repository. We will make use of five atomic component designs to model the component-based *MR* system in X-MAN (as shown in Fig. 3). These design components are *Tank*, *Controller*, *Evaluator*, and *PrAbort*, and *PrIdling*. *PrAbort* models the computation when an operation mode of the ground fuel transfer system aborts. Similarly, *PrIdling* encapsulates the computation when an operation becomes ilde. The final 2 components are relatively simple in our case study, basically returning the output of the component with some constant values, so, we omit their description here, and concentrate on the remaining 3 components.

- Tank: This component maintains some stored state variables which represent a mass of fuel in a particular tank. Fuel output and inlet are modelled by the setting of arguments corresponding to each of 3 ports. A postive mass represents fuel input to the tank, negative signifies outputs. The tank component contains an internally selected maximum-mass figure. Input of a fuel mass which causes this figure to be exceeded results in an overflow mass being outputted. The component also outputs the current mass, states and status of pumps and valves of each tank.
- Controller: calculates the fuel flow between four instances of the Tank component, modelling the functionality of the pumps, valves and pipes linking them. The outputs of each tank along with the command signals for each pump and valve are fed to this component.
- Evaluator: accepts the system inputs and evaluates the right operation mode based on the inputs. The evaluation logic is represented as the first-order logic formula in this case. It outputs the evaluation result.

X-MAN model system shown in Figure 3 makes use of 8 atomic component instances, and 5 composite components, involving three different kinds of composition connectors – *Pipe*, *Selector* and *Sequencer*. Firstly, we compose four instances of the design component *Tank*: *LTank*, *CTank*, *RTank*, *STank*, which model *LWT*, *CWT*, *RWT* and *SP* respectively, into a composite component named *TanksReader* using the *Sequencer* connector *Seq_reader*. In the same way, these four component instances are composed together into another composite component named as *TanksWriter* using the connector *Seq_writer*. Thus, every *tank* instance is accessed twice for each iteration of the model. Next, *TanksReader*, an instance of *Controller* named as *CTRL*, and *TanksWriter* are further composed together into a single larger composite component *In_Process_Pipe* by a *Pipe* connector *Pipe1*, which models the *In_Process* operation. Then, *In_Process_Pipe* is composed with the component *A*, which is the instantiation of the design *PrAbort*, and the component *I*, which is the intantiation of *PrIdle*, by the *Selector* connector *Sel0* to construct the more complex composite component *MR_Process*. Finally, by means of another *Pipe* connector *Pipe0*, *MR_Process* is composed with the component *EV*, which is an instantiation of the design component *Evaluator*.

The model behaves as follows at run-time: when a call (with inputs) reaches the top-level connector *Pipe0*, the connector first calls the atomic component *EV* with its required inputs and passes the outputs to the *Sel0* connector. Based on the values passed, *Sel0* evaluates the selection condition in order to choose among the composite components *In_Process_Pipe*, *A* or *I*. If *In_Process_Pipe* is chosen, *TanksReader* is first called and its monitoring outputs are passed to *CTRL*, then finally the computed command ouputs of *CTRL* are fed into *TanksWriter*. When either *TanksReader* or *TanksWriter* is executed, the *LTank*, *CTank*, *RTank*, *STank* are accessed in sequence, but the usage of the outputs from these *Tank* instances is different. For instance, *TanksReader* manipulates the outputs which represent the current mass, the state and status of the tank, while *TanksWriter* uses those outputs that correspond to the handlers of the pump and valve of each tank. If *A* is chosen, the model starts running as the *Abort* operation mode in *MR*. Otherwise, *I* is chosen to simulate the *Idle* behaviour of *MR*.

C. Component-based Verification of the Manual Refuel Example

Consider the atomic component *LTank* of the *MR* modelling system. It contains 6 contracts: 3 of these specify the integer ranges of the outputs parameters, the other 2 describe the conditions under which fuel may be added without overflow occuring, and the final contract relates to the minimum and maximum execution time of the component. We use one contract *cMassRange* for the component *LTank* as an example:

Contract($cMassRange$) {
  Inputs: $iAddedMass : int$
  Ouputs: $oMass : int$, $oMaxMass : int$
  Pre-condition: $true$
  Post-condition: $oMass \geq 0 \wedge$
  $oMass \leq oMaxMass$
}

Two integer outputs are used in this contract. *oMass* reflects the current fuel mass of the tank, and *oMaxMass* represents the internally selected maximum mass figure. The *Pre-condition* of this contract is *true* which means no assumption of the component inputs is made. The corresponding *Post-condition* requires that *oMass* must always be a positive value and *oMass* must never exceed the value of *oMaxMass* after the execution of the *LTank* component. As a further example, the contract for specifying the extra-functional timing property of this component uses the special parameters *preTime* and *postTime*, which are globally accessible throughout the component, to describe the timing constraint as follows.

Contract($cTankTiming$) {
  Inputs:
  Ouputs:
  Pre-condition: $true$
  Post-condition: $postTime \leq preTime + 5 \wedge$
  $postTime \geq preTime + 1$
}

According to this contract, the execution time (in seconds) of this contract must always fall within the range $[1, 5]$ under any combinations of inputs. In addition to the basic *Range-Contract* of output variables and *timing-Contract* as introduced above, some component contracts formulate the complex data constraints between model inputs and outputs that are derived from the system requirement. For instance, the contract for the *Controller* component instance $CRTL$ is as follows:

Contract($cFuelConservationController$) {
  Inputs: $iMass[4] : int$,
  Ouputs: $oAddedMass[4] : int$
  Pre-condition: $iMass[0] \geq 0 \wedge iMass[1] \geq 0 \wedge$
  $iMass[2] \geq 0 \wedge iMass[3] \geq 0$
  Post-condition: $oAddedMass[0] +$
  $oAddedMass[1] +$
  $oAddedMass[2] +$
  $oAddedMass[3] = 0$
}

In this contract for *CTRL*, the input parameter $iMass$ is an Integer array which is passed from the composite component *Seq_reader*. Each element corresponds to the output parameter $oMass$ of *LTank*, *CTank*,*RTank*,*STank* respectively. The output parameter $oAddedMass$ is also an Integer array which is passed to the composite component *Seq_writer*. And each element corresponds to the input parameter $iAddedMass$ of four *Tank* component instances. This *Post-condtion* of this contract requires that the fuel mass must be conserved in the *CTRL*.

Vertical Verification Phase: We apply the software Model Checker *CBMC* to formally verify that the implementation of every atomic component in *MR* (8 components in total) satisfies its corresponding contracts. For instance, we check that the C implementation of the component $CTRL$ satisfies the post-condition specified in the contract *cFuelConservationController* given above, under the assumption that the input parameters adhere to the *Pre-condition* constraint. Our verification tool first automatically instruments the following assumption statement at the beginning of the implementation code.

assume($cFuelConservation.Pre - condition$);

and, the following assertion at the end of the code.

assert($cFuelConservation.Post - condition$);

Then, *CBMC* is applied to verify the instrumented code.

Due to the exhaustive search within the unwinding depth and precise modeling of data variables, *CBMC* can detect non-trivial corner case bugs, which eluded discovery during testing by means of a conventional test-suite derived from the requirements. As an example, consider the following code fragment from the *CTRL* implementation where the variables have been renamed.

const int $sink\_num = 2; //****$
...
int $val\_a = a/sink\_num; *$
int $val\_a\_remainder = a\%b;$
...
if$(v\_open)\{$
  $val\_c = val\_a;$
}
else{
  $val\_c = 0;$
}
...
if$(val\_c > 0)\{$
  $val\_c = val\_c + val\_a\_remainder;$
}

...

The variable a models a total amount of flow that is distributed between a given number of sinks. The amount is given as an integer quantity, and there can therefore be a remainder (val_a_remainder), which is to be apportioned to the valve flow val_c. Consider the special execution trace when

$$a = 1;\ v\_open = 1;$$

Then,

$$val\_a = 0;\ val\_a\_remainder = 1;\ val\_c = 0;$$

i.e., the remainder is non-zero but is not apportioned to the valve flow val_c. The buggy value of val_c propagates to the outputs and causes the violation of the *Post-condition* of *cFuelConservation*. The variables a and v_open are both internal variables. It is difficult for a conventional testing technique to find the primary inputs stimuli that could observe this corner case scenario where a is set to 1 and v_open is set to be *True* at the same time. Consequently, the error was missed by the conventional test suite, while *CBMC* quickly identifies an erroneous execution trace as above. Analyses revealed that the bug occurred because the code was adopted from a similar algorithm using floating-point arithmetic, and did not handle integer division truncation properly.

Horizontal Verification Phase: After the contracts of all components have been verified, we can start horizontal verification, which checks the properties of the modeling system based on the verified contracts of the components and their composite relationships. For instance, given a system C composed of two verified components A and B connected by a Selector, we combine the post-conditions in the contracts of A and B as follows:

$$f = (\textit{selC-cond} \wedge post_A) \vee (\neg \textit{selC-cond} \wedge post_B)$$

where $post_A$ and $post_B$ denote the post-conditions of components A and B, respectively. *selC-cond* is the condition that the sub-component is chosen in the case of using Selector connector. In this example, if *selC-cond* is true, component A is selected for execution; otherwise, component B is selected. For the composite systems using other connectors, the X-MAN-Verifier can also automatically derive the formula f from the proved contracts of sub-components, according to the composition behavious of the corresponding connectors with respect to the properties under verification.

Then, the X-MAN-Verifier checks whether f implies $post_C$. If so, we have proven that the component C satisfies its contract; otherwise the contract may or may not hold. The X-MAN-Verifier can produce a trace that demonstrates how the component C fails to respect its contract. For a given timing property

$$postTime < preTime + 55$$

for *MR*, the X-MAN-Verifier returns *hold*. On the other hand, the timing property

$$postTime < preTime + 50$$

may be violated and the tool provides a counterexample. We can conclude that the *MR* system is guaranteed to finish execution within 55 time steps, but may exceed 50 time steps. The counterexample extracted is as follows:

$Inputs$:
  $in0 = 0, ..., in3 = 0;$
  $preTime = 0;$
$- - - - - - - - - - - - -$
$CallingEC$
$Inputs$:
  $preTime = 0,$
$Outputs$:
  $value = 1;$
  $postTime = 5;$
$- - - - - - - - - - - - -$
$CallingLA$
....
$- - - - - - - - - - - - -$
$Outputs$:
  $out0 = 1, ..., out4 = 1;$
  $postTime = 53;$

This trace shows the system inputs and outputs, $preTime$ and $postTime$ of *MR*, and the components on the execution trace with their inputs, ouputs, $preTime$ and $postTime$.

VI. Conclusions

In this paper, we proposed a new compositional design and verification methodology based on X-MAN using formal methods, and investigate its applicability to the safety-critical embedded systems using a case study implementing the ground fueling scenario of an aircraft.

Future work includes the evaluation of the framework using models that utilize the support for IEEE floating-point arithmetic [6]. Further research will include the development of test-suite generation algorithms that exploit the strong isolation of the components and the information about the structure of their composition, e.g., pursuing ideas similar to those that have been applied to Simulink dataflow diagrams [14].

References

[1] A. Basu, S. Bensalem, M. Bozga, J. Combaz, M. Jaber, T.-H. Nguyen, and J. Sifakis. Rigorous component-based system design using the BIP framework. *IEEE Software*, 28(3):41–48, 2011.

[2] A. Basu, M. Bozga, and J. Sifakis. Modeling heterogeneous real-time components in BIP. In *SEFM*, pages 3–12, 2006.

[3] A. Basu, M. Gallien, C. Lesire, T.-H. Nguyen, S. Bensalem, F. Ingrand, and J. Sifakis. Incremental component-based construction and verification of a robotic system. In *ECAI*, pages 631–635, 2008.

[4] S. Bensalem, M. Bozga, T.-H. Nguyen, and J. Sifakis. Compositional verification for component-based systems and application. *IET Software*, 4(3):181–193, 2010.

[5] S. Bliudze and J. Sifakis. Causal semantics for the algebra of connectors. *Formal Methods in System Design*, 36(2):167–194, 2010.

[6] A. Brillout, D. Kroening, and T. Wahl. Mixed abstractions for floating-point arithmetic. In *Formal Methods in Computer-Aided Design (FMCAD)*, pages 69–76. IEEE, 2009.

[7] S. Chaki, J. Ivers, N. Sharygina, and K. C. Wallnau. The ComFoRT reasoning framework. In *Computer Aided Verification (CAV)*, volume 3576 of *LNCS*, pages 164–169. Springer, 2005.

[8] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *Proc. of TACAS*, 2004.

[9] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.

[10] I. Crnkovic, S. Larsson, and M. Chaudron. Component-based development process and component lifecycle. *Journal of Computing and Information Technology*, 13(4):321–327, 2005.

[11] L. de Alfaro and T. Henzinger. Interface theories for component-based design. In *Embedded Software (EMSOFT)*, volume 2211 of *LNCS*, pages 148–165. Springer, 2001.

[12] S. Fleury, M. Herrb, and R. Chatila. Genom: A tool for the specification and the implementation of operating modules in a distributed robot architecture. In *In International Conference on Intelligent Robots and Systems*, pages 842–848, 1997.

[13] D. Giannakopoulou, C. Păsăreanu, and H. Barringer. Assumption generation for software component verification. In *ASE*, pages 3–12. ACM, 2002.

[14] N. He, P. Rümmer, and D. Kroening. Test-case generation for embedded Simulink via formal concept analysis. In *Design Automation Conference (DAC)*, pages 224–229. ACM, 2011.

[15] D. Kroening and O. Strichman. *Decision Procedures – An Algorithmic Point of View*. Springer (Theoretical Computer Science series), 2008.

[16] K.-K. Lau, P. V. Elizondo, and Z. Wang. Exogenous connectors for software components. In *Proc. 8th Int. Symp. on Component-based Software Engineering*, pages 90–106. Springer, 2005.

[17] K.-K. Lau, M. Ornaghi, and Z. Wang. A software component model and its preliminary formalisation. In *Proc. 4th International Symposium on Formal Methods for Components and Objects*, volume 4111 of *LNCS*, pages 1–21. Springer, 2006.

[18] B. Meyer. Contracts for components. *Software Development*, 8(7):51–53, 2000.

[19] S. Quinton and S. Graf. Contract-based verification of hierarchical systems of components. In *Intl. Conference on Software Engineering and Formal Methods (SEFM)*, pages 377–381. IEEE, 2008.

[20] P. Rümmer. A constraint sequent calculus for first-order logic with linear integer arithmetic. In *LPAR*, volume 5330 of *LNCS*, pages 274–289. Springer, 2008.

[21] R. Smith, B. Meyer, C. Szyperski, and G. Pour. Component-based development? Refining the blueprint. *Technology of Object-Oriented Languages, International Conference on*, 0:563, 2000.

[22] K. Wallnau. A technology for predictable assembly from certifiable components (PACC). In *Technical Report CMU/SEI-2003-TR-009, SEI, CMU*, 2003.

APPENDIX

Here we show some screenshots of the X-MAN tool during the design and verification of the Ground Fuel Management System.

Figure 4 shows the design of an atomic component.

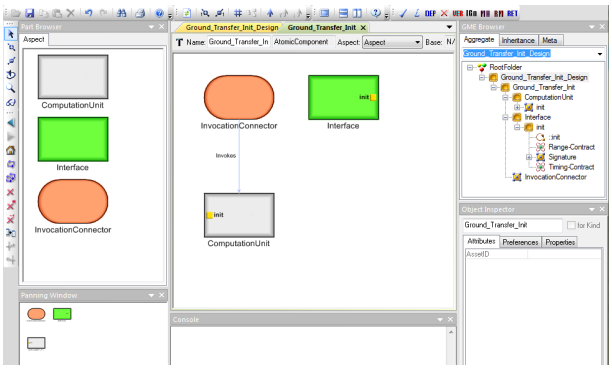


Figure 4. Component design.

Figure 5 shows the repository of (atomic) components that have been built.

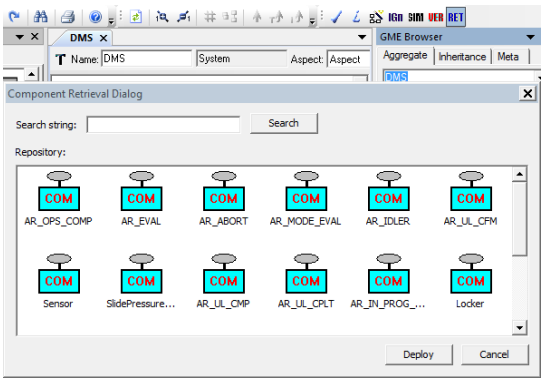


Figure 5. Component repository.

Figure 6 shows the design of the system, using pre-defined (atomic) components from the repository.

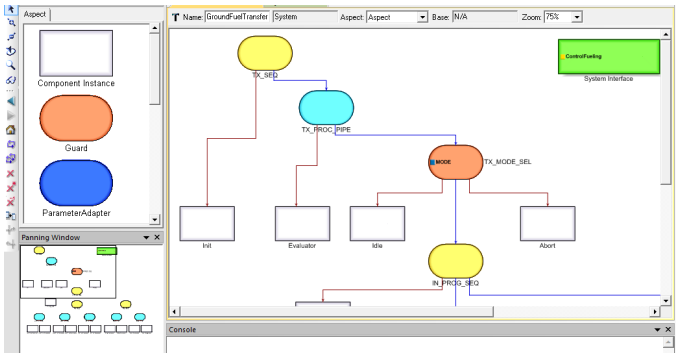


Figure 6. System assembly from repository components.

Figure 7 shows the verifier tool in action, verifying the contract of an atomic component.

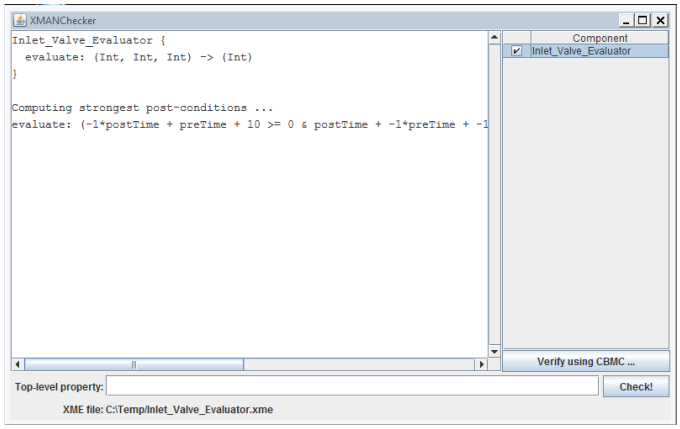


Figure 7. Component verification.

Figure 8 shows the verifier tool in action, verifying a contract at system level.

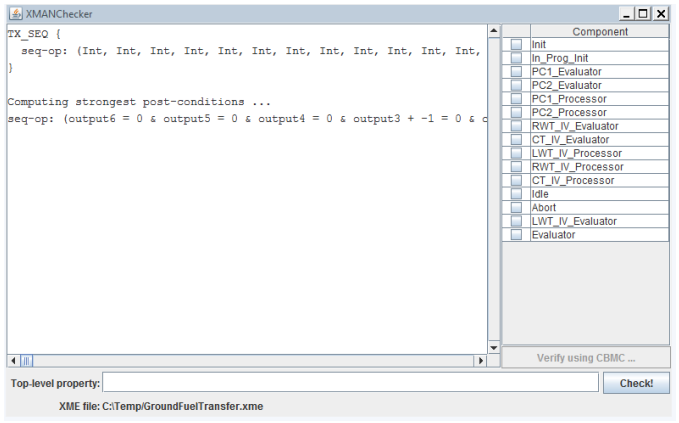


Figure 8. System verification.