TRACE: Toolkit for Requirements Analysis, Capture, and Elicitation

Sarat Chandra Varanasi(✉), Baoluo Meng, Robert Lorch, Abha Moitra, Kit Siu, Saswata Paul, Michael Durling, Neha Beniwal, and Nikita Visnevski

GE Aerospace Research, Niskayuna, NY, USA
{saratchandra.varanasi,baoluo.meng,robert.lorch,abha.moitra,siu,saswata.paul,durling,neha.beniwal,nikita.visnevski}@geaerospace.com

Abstract. Verification and validation of requirements is a crucial step in Model-Based Engineering to avoid costly re-work involved in the re-design and re-validation of systems and software due to flawed requirements. Verification of requirements necessitates capturing them in a formal notation that is amenable to analysis by formal methods tools. Further, realistic systems specify requirements cutting across subsystems that involves multi-physics interactions of subsystems' properties of interest. Such multi-domain requirements also constrain the design space of sub-systems. We present TRACE (Toolkit for Requirements Analysis, Capture, and Elicitation) for systems that allows for the capture of different types of requirements. TRACE facilitates the analysis of system requirements including a wide variety of analyses such as conflict, independence, contingency, and completeness analyses available in several practical requirements analyses tools. In addition, requirements regarding component properties may also be formalized as guarantees and checked for implication by other requirements using backend Satisfiability Modulo Theories (SMT) solvers. Detailed diagnostic information for failed guarantees, including `UNSAT` cores and counterexamples, will be provided to assist users in pinpointing and resolving the identified issues. We apply TRACE to an industrial Hybrid Electric Propulsion System (HEPS) use case, showcasing various capabilities.

1 Introduction

Verification and Validation in systems engineering begins with crucial step of requirements capture system design, to avoid costly re-work in its development lifecycle due to incorrect and ambiguous requirements discovered later. More often, the system requirements involve the declaration of a system's desired operating constraints that span multiple domains in addition to low-level desired system functional behaviors. Traditionally, low-level functional requirements amenable to formal analyses have been captured in several formalisms [3,10].

S. C. Varanasi and B. Meng—These authors contributed equally to this work.

A. Dutle et al. (Eds.): NFM 2025, LNCS 15682, pp. 380–399, 2025.
https://doi.org/10.1007/978-3-031-93706-4_22

In this paper, we introduce TRACE (Toolkit for Requirements Analysis, Capture, and Elicitation), which allows for the capture and analyses of different types of system requirements that span multiple domains in addition to low-level functional requirements. TRACE is integrated into an Eclipse-based IDE, TRACE IDE, that allows developers and engineers to author the requirements and provides feedback on syntactic and semantic issues as the requirements are authored. The requirements in TRACE can be formally analyzed using SMT solvers supporting a range a requirement analyses. In particular, TRACE IDE performs the following checks: conflict, contingency, independence, completeness, and guarantee validity. TRACE's SMT solver backend is completely transparent to the users, and the IDE provides significant tool support for easy interpretability of the requirements analyses results. We believe that using an SMT solver backend will help TRACE leverage continuously improving solver enhancements.

An important feature of TRACE involves the declaration of units which is crucial when authoring a large number of requirements with multiple sub-system interactions involving different physical properties. As stated before, an important aspect of complex systems is that they typically span a number of domains. For example, an aircraft engine needs to incorporate electrical, propulsion, heat dissipation, etc. as well as the dependencies between these domains. For instance, heat generated by a propulsion system must be rejected sufficiently to maintain temperatures within its permissible operating condition. TRACE allows domains to be declared to support the capture and analysis of such multi-domain requirements. Further, TRACE admits the capture of domain-specific equations, for instance the equations that involve the calculation of power dissipated by a battery given its voltage and current ratings.

We describe the TRACE language and IDE and showcase their application on an industrial use case of capturing some requirements for a parallel hybrid electrical propulsion system (HEPS). In recent decades, electrified aircraft propulsion (EAP) has emerged as a vital solution in reducing carbon dioxide emissions and improving aircraft efficiency. Among the various EAP systems, parallel HEPS makes it feasible to target national/international aircraft markets with distance ranges greater than 3000 mi by maintaining optimum weight and providing fuel reductions [23,37]. A parallel HEPS is targeted to provide a more efficient propulsion method as compared to traditional engines by using electrical power in addition to mechanical power delivered from a conventional gas turbine during aircraft takeoff and climb. In this paper, we apply TRACE to a parallel HEPS use case with requirements spanning thermal and electrical domains. Our key contributions include:

- A novel language and tool – TRACE, for capturing and analyzing formal system requirements
- A formal description of TRACE syntax, semantics, and unit checking
- A formal description of TRACE analysis capabilities
- A thorough evaluation of TRACE's capabilities on an industrial hybrid electric propulsion system (HEPS) use case.

2 TRACE Requirements

iption of the TRACE language is provided in Appendix 3. In this section, we provide a brief description of the kinds of requirements supported in TRACE.

T1 Ubiquitous requirements - that declare that a system shall take a specified action unconditionally.

```
System shall set fault to 1.0
```

T2 Condition-Action requirements - that specify system behavior in response to a condition. Condition-Action requirements are the most common type of functional requirements captured in real industrial projects. Such requirements involve the use of *monitored* and *controlled* variables. Monitored variables of a requirement specify system conditions that determine system behavior, as specified by the values that its controlled variables take, in response to its monitored variable conditions.

```
System shall set controlled_variable to 1.0 when monitored_variable is greater than 2.0
```

T3 High-level requirements - that capture high-level system properties that should not be violated. High-level requirements are decomposed into low-level requirements which may span multiple physics domains.

```
Aircraft shall provide motor_power >= 500 kW.
```

T4 Equations - specify physics-based calculations of various system properties with appropriate units. A simple battery power calculation is a product of battery voltage and current.

```
Equation Battery::battery_power = (battery_voltage * battery_current) kW.
```

T5 Multi-domain requirements - similar to requirement types *T1* through *T3*, except, that they refer to components and variables across multiple domains.

```
BatteryDomain domain batteryDomain {
  Requirement BatteryPower: Battery shall provide battery_power >= 200 kW. }
PowerElectronicsDomain domain powerElectronicsDomain{
  Requirement PowerConverterEfficiency:
    PowerConverter shall provide efficiency >= 0.96 and efficiency < 1.
  Equation PowerConverterLoss:
    PowerConverter shall provide loss= (1 - efficiency) * Battery:battery_power kW.
  Requirement PowerConverterLossThreshold: PowerConverter shall provide loss <= 100 kW.}
```

Note that the equation `PowerConverterLoss` involves the `BatteryDomain` & `PowerElectronicsDomain` whereas `PowerConverterEfficiency` involves only the `PowerElectronicsDomain`.

T6 Guarantees - specify that low-level requirements and equations logically imply high-level requirements. For example, requirements `BatteryPower`, `PowerConverterEfficiency`, and the equation `PowerConverterLoss` imply `PowerConverterLossThreshold`. And this decomposition can be captured as a *guarantee* statement and checked in TRACE. Guarantee G1: BatteryPower, PowerConverterEfficiency, PowerConverterLoss, PowerConverterEfficiency imply PowerConverterLossThreshold.

```
Guarantee G1: BatteryPower, PowerConverterEfficiency, PowerConverterLoss,
     PowerConverterEfficiency imply PowerConverterLossThreshold.
```

3 TRACE Language

TRACE Syntax and Semantics. A system *requirement* describes an intended behavior or property of the system. The TRACE language specifies constructs for the declaration and definition of *Packages, Component Types, Subtyping Component Types, Component Type Variables, Enumerated Datatype Definitions, Record Datatype Definitions, Units Definitions, Properties, Equations, & Constants* which serve as the building blocks upon which several different types of requirements can be captured.

A system description in TRACE comprises of *Component type definitions* that can be *Sub-typed*. Each component type contains *Variables*, each of which describe a potential logical state or a property of the system. Each variable has an associated *Datatype* and may represent either a monitored or controlled variable of some requirement definition in which that particular variable is used to represent a logical state of the system. A component can have a variable annotated as a *Property* which differentiates it from a monitored/controlled connotation. A variable that is identified as a property is used in physics-based equations that characterize the input-output behavior of (sub-)systems at the physical level. These properties become essential when capturing high-level requirements involving multiple physics domains. Building-block features for a system description are illustrated by examples.

```
PowerConversionController is a component type with variables {
fault : real, power_output : real as a property }
```

TRACE provides modularity of definitions using packages and import mechanisms similar to most modern programming languages and domain-specific languages (DSLs). Packages essentially serve as namespaces and imports can be specified similar to imports in Java.

```
package ControllerRequirements.
import Definitions.PowerConversionController;
import Definitions.*;
```

In addition to specifying data types for variables, TRACE provides support for defining units and specifying units for variables. Units, where appropriate and needed, ensure that the requirements engineer will be completely aware of how the variables with units are used within equations and arithmetic expressions. Units become very important when variables that represent physical quantities need to be captured precisely against the physics-units they are measured within, in equations and formulas. TRACE reports violations in dimensionality of the units used in arithmetic expressions and equations, which is paramount to an engineer when authoring requirements.

```
Units { kg : "kilogram", A : "ampere", V : "volt" }
DCDCController is a component type with variables { voltage : real with unit V }
```

TRACE supports subtyping component types which significantly improves the reusability of component variables when authoring requirements. In TRACE, a

component type can be a subtype of another component type and extend the parent type with its own variables modulo name clashes. Subtypes in TRACE can be extended to arbitrary levels while restricting subtyping to a single parent.

```
DCACController is a type of PowerConversionController with variables {
 ac_current_channel_a : real with unit A, ac_current_channel_b : real with unit A }
```

TRACE allows the definition of enumerated data types and record types common to most modern programming languages and DSLs.

```
CockpitControl is an enum type with values [Cockpit_Controls, Automated_Control]
Channel is a record type { raw_value : real, converted_value : real }
```

TRACE allows for the declaration of domain types which are important for analyzing requirements of (sub-)systems spanning different domains. For example, requirements can be specified comprising of thermal phenomena related to the power characteristics of an electrical system.

```
WeightRollup, PowerElectronics are domain types.
```

Expressions in TRACE. Formulas and expressions specified in requirements are constructed using terms and operators. Ill-typed expressions and formulas will be flagged as errors.

- Logical formulas & constants: conjunctions (`and`), disjunctions(`or`), `true`, `false`
- Comparison formulas: is equal to (=), is less than (<), is equal to or less than (<=), is greater than (>), is equal to or greater than (>=),
- Arithmetic expressions: add (+), sub (−), mul (*), and div (/)
- Arithmetic constants: integer and real numbers

TRACE Requirements Semantics. We introduce a few notations and relations. Let C denote a component type, and $\mathcal{I}(C)$ denotes the set of instances of component type C including its subtypes. We use $\mathcal{I}(C)$ to ground the requirements over a component type's instances. If no instance is specified for a component then we assume $\mathcal{I}(C) = \{C\}$. If C' is an instance component type C, then $\mathcal{I}(C') = \{C'\}$. Grounding over arithmetic expressions involves generating all possible expression instances by substituting all instances of a component type's variables present in the expression. More precisely, $\mathcal{I}(constExpr) = \{constExpr\}$ and $\mathcal{I}(S.x \otimes S'.y) = \{s.x \otimes s'.y | s \in \mathcal{I}(S) \wedge s' \in \mathcal{I}(S')\}$ where $\otimes$ is one of the arithmetic or inequality operators: $\{+, \times, -, /, <, >, \leq, \geq\}$. Requirements are fundamentally assertions, and their semantics are captured as logical statements, specifically into propositions. Before we delve into their SMT-LIB translation, we provide their denotational semantics at a propositional level. Semantics of arithmetic operations, arithmetic equalities and inequalities, and enumeration equalities and inequalities carry their usual meaning and are not elucidated. Semantics of functional requirements:

$$[\![S \textit{ shall set } c \textit{ to } expr]\!] \triangleq \{(s.c = [\![expr]\!]) \mid s \in \mathcal{I}(S)\}$$

$[\![S \textit{ shall set } c \textit{ to } v \textit{ when } m_1 = e_1 \otimes_1, .., \otimes_n m_n = e_n]\!] \triangleq \{((m_1 = [\![e_1]\!] \otimes_1, .., \otimes_n m_n = [\![e_n]\!]) \Rightarrow (s.c = [\![v]\!]))| s \in \mathcal{I}(S) \wedge \otimes_i \in \{\vee, \wedge\} \wedge 1 \leq i \leq n\}$

In general, a requirement's assertions are grounded over all possible variables belonging to various instances of a component type that it references.

$[\![S \textit{ shall set } S'.c \textit{ to } S''.val \textit{ when } S'''.m = expr]\!] \triangleq \{\bigwedge((s'''.m = [\![e']\!] \Rightarrow (s'.c = [\![s''.val]\!]))| \; s'' \in \mathcal{I}(S''), s''' \in \mathcal{I}(S'''), e' \in \mathcal{I}(expr) \; \}$ The general case follows similarly when there are multiple monitored variable conditions expressed in a requirement. Semantics of high-level requirements are similar to functional requirements but they are subject to different analyses.

Table 1. TRACE type checking rules.

$$\frac{}{true : boolean}(\text{true}) \qquad \frac{}{false : boolean}(\text{false}) \qquad \frac{}{[0-9] + (.[0-9]*)? : real}(\text{number})$$

$$\frac{E \textit{ is an enum type with values} \{v_1, v_2, ..., v_n\}}{E.v_1 : enum \qquad E.v2 : enum \qquad ... \qquad E.v_n : enum}(\text{enum})$$

$$\frac{R \textit{ is a record type } \{x_1 : \tau_1, x_2 : \tau_2, ..., x_n : \tau_n\}}{R.x_1 : \tau_1 \qquad R.x_2 : \tau_2 \qquad ... \qquad R.x_n : \tau_n}(\text{record})$$

$$\frac{C \textit{ is a component type with variables} \{x_1 : \tau_1, x_2 : \tau_2, ..., x_n \tau_n\}}{C.x_1 : \tau_1 \qquad C.x_2 : \tau_2 \qquad ... \qquad C.x_n : \tau_n}(\text{comp-var})$$

$$\frac{\Gamma \vdash x : real \qquad \Gamma \vdash y : real}{\Gamma \vdash x \; ArithOp \; y : real}(\text{arith-construct})$$

$$\frac{\Gamma \vdash x : \tau_x \qquad \Gamma \vdash y : \tau_y \qquad (\tau_x \neq real \vee \tau_y \neq real)}{\Gamma \vdash x \; ArithOp \; y : error}(\text{arith-error})$$

$$\frac{\Gamma \vdash x : \tau_x \qquad \Gamma \vdash y : \tau_y \qquad \tau_x \neq \tau_y}{\Gamma x \; CompOp \; y : error}(\text{comp-error})$$

$$\frac{\Gamma \vdash x_1 : \tau_1, x_2 : \tau_2, ..., x_n : \tau_n \qquad \exists i \wedge 1 \leq i \leq n \wedge \tau_i \neq real}{\Gamma \vdash ArithExpr(x_1, x_2, ..., x_n) : error}(\text{arith-expr-error})$$

$$\frac{\Gamma \vdash x : \tau_x, expr : \tau_e \qquad \tau_x \neq real \vee \tau_e \neq real}{\Gamma \vdash x = expr : error}(\text{eq-error})$$

$$\frac{\Gamma \vdash x : \tau_x, expr : \tau_e \qquad \tau_x \neq \tau_e}{\Gamma \vdash <S> \textit{ shall set } x \textit{ to } expr : error}(\text{req-error-1})$$

$$\frac{\Gamma \vdash x : \tau_x, expr : \tau_e \qquad \tau_x \neq \tau_e}{\Gamma \vdash <S> \;\; shall \; provide \; x \; RelOp \; expr \;\; <Unit> : error}(\text{req-error-2})$$

TRACE Expression Type-Checking and Unit-Checking. TRACE reports type errors back to the requirements engineer. The TRACE type-checker considers the following abstract types when looking for type errors in expressions: *real, boolean, enum, record, error*. Any expressions whose type results in *error* type is reported back to the user with an explanation triggering that error. Table 1 summarizes the type- checking rules for TRACE, where Γ denotes the context under which type inference is performed and it is standard notation for typing judgments for expressions. Unit checking in TRACE involves checking the dimensionality of units present in an expression. We only check dimensionality

violations with respect to a single unit in an expression involving arithmetic equality and inequality operators. Let $\Delta_\mu(expr)$ denote the dimensionality of unit μ in the arithmetic expression $expr$. If $\Delta_\mu(expr) = \perp$, then the dimensionality is either erroneous or undefined. Table 2 summarizes unit checking rules to determine the computation of dimensionality of units.

Table 2. TRACE unit checking rules.

$$\frac{C \textit{ is a component type with variables } \{x_1 : \tau_1 \textit{with unit } \mu_1\}}{\Delta_{\mu_1}(C.x_1) = 1 \qquad \Delta_{\mu \neq \mu_1} C.x_1 = 0} \text{(dim-atomic)}$$

$$\frac{\Gamma \vdash \Delta_\mu(x) = m \qquad \Gamma \vdash \Delta_\mu(y) = n}{\Gamma \vdash \Delta_\mu(x + y) = max(m, n)} \text{(sum-unit)} \qquad \frac{\Gamma \vdash \Delta_\mu(x) = m \qquad \Gamma \vdash \Delta_\mu(y) = n}{\Gamma \vdash \Delta_\mu(x - y) = max(m, n)} \text{(diff-unit)}$$

$$\frac{\Gamma \vdash \Delta_\mu(x) = m \qquad \Gamma \vdash \Delta_\mu(y) = n \qquad m \neq n}{\Gamma \vdash \Delta_\mu(x \; CompOp \; y) = \perp} \text{(unit-error)}$$

4 TRACE Analyses

We translate the requirements to SMT while using the denotational semantics presented in the previous section. With the formal representation, we perform a wide variety of analyses using SMT solvers. Below, we discuss the analyses by outlining the satisfiability queries sent to the SMT solver for analysis. First, we cover some preliminaries. An SMT solver checks for the *satisfiability* of an input set of first-order formulae with respect to a set of background theories, e.g., integer arithmetic; in other words, it looks for an assignment of variables to values (*i.e.*, an *interpretation*) such that all input formulae evaluate to true. A formula is *valid* if it is true in every interpretation. Checking the formula φ for validity can be reduced to checking the satisfiability of $\neg\varphi$ – if $\neg\varphi$ is *un*satisfiable, then φ is valid. A formula is *falsifiable* if there exists some interpretation such that the formula evaluates to false. Checking the formula φ for falsifiability can be reduced to checking the satisfiability of $\neg\varphi$ – if $\neg\varphi$ is satisfiable, then φ is falsifiable.

Pairwise Conflict Analysis. It checks whether any pair of requirements contains a logical contradiction. Suppose we have two requirements $\alpha_1 \Rightarrow \beta_1$ and $\alpha_2 \Rightarrow \beta_2$. To check for a pairwise conflict, we check the satisfiability of two formulae, $\alpha_1 \wedge \beta_1 \wedge (\alpha_2 \Rightarrow \beta_2)$ and $\alpha_2 \wedge \beta_2 \wedge (\alpha_1 \Rightarrow \beta_1)$ in separate SMT queries. Conceptually, we can think of each of these formulae as taking the conjunction of the two requirements, except in the first we assume that the first antecedent (α_1) holds and in the second we assume that the second antecedent (α_2) holds. If either formula is unsatisfiable, then there is a logical contradiction between the two formulae, as the tool has determined that it is impossible for both requirements to simultaneously hold. If both formulae are satisfiable, then the check passes. If we do not assume that one of the antecedents holds, then in many cases the conjunction of the two formulae can be trivially satisfied by falsifying both α_1 and α_2.

Pairwise Independence Analysis. It checks whether any requirement is made redundant by another requirement. For every two requirements φ_1 and φ_2, we check the validity of $\varphi_1 \Rightarrow \varphi_2$ and $\varphi_2 \Rightarrow \varphi_1$ in separate SMT queries. If either formula is valid, then one of the requirements must be strictly weaker than another, and the check fails. Otherwise, the check succeeds.

Contingency Analysis. It checks that every requirement φ is both satisfiable and falsifiable. It is important for each individual requirement to be satisfiable because we want it to be possible to implement the requirement. In addition, we want the requirement to be falsifiable—if not, the requirement trivally holds for any possible implementation. If φ is indeed both satisfiable and falsifiable, the check succeeds. Otherwise, the check fails.

Global Contingency Analysis. It is the same as contingency analysis, except for the conjunction of *all* requirements. In other words, global contingency analysis checks the satisfiability and falsifiability of $\varphi_1 \wedge \cdots \wedge \varphi_n$, for n requirements.

Completeness Analysis. It checks that for every possible assignment of monitored variables to values, the values of the controlled variables are uniquely specified. If they are not, then the behavior of at least one of the controlled variables is underspecified by the requirements. To perform this check, we check the satisfiability of the conjunction of the set of requirements as well as a set of *primed* versions of all requirements. In other words, we check the satisfiability of $\varphi_1 \wedge \cdots \wedge \varphi_n \wedge \varphi_1{}' \wedge \cdots \wedge \varphi_n{}'$. Each primed formula φ' represents the same formula as φ, except each variable is replaced with a primed copy. Additionally, we add the constraint that each monitored variable mv is equal to its primed counterpart ($mv = mv'$), and also that at least one controlled variable cv_i is distinct from its primed counterpart ($cv_1 \neq cv_1{}' \vee \cdots \vee cv_m \neq cv_m{}'$). If this set of formulae is satisfiable, then there exist two assignments (to the primed and unprimed variables) that satisfy all requirements such that the primed and unprimed monitored variables are all equal, but at least one primed controlled variable differs from its unprimed counterpart. These two assignments prove that there is some underspecified controlled variable that can take on multiple values under the same assignment of monitored variables (while still respecting the requirements). Hence, the check fails. If the set of formulae is unsatisfiable, then the check succeeds.

Guarantee Analysis. Guarantee analysis is to establish the validity of logical implication of a set of requirements and equations decomposing another high-level requirement. For a guarantee $g : r_1, ..., r_n$ *imply* R, checking the validity of g can be reduced to a logical problem: checking the validity of $\alpha_{r1} \wedge \alpha_{r2} ... \wedge \alpha_{rn} \implies \beta_R$, where $\alpha_{r1}, ..., \alpha_{rn}, \beta_R$ are logical representations of corresponding requirements, $\alpha_{r1}, ...\alpha_{rn}$ are called antecedents, and β_R is called the consequent. Prior to checking the validity of guarantees, it is necessary to ensure the consistency of the prerequisites outlined in the antecedent of guarantees. Inconsistencies within the antecedent would render it false, thereby causing the predicament wherein falsity implies any proposition, trivially establishing the validity of guarantees. When the requirements on the antecedent side are found

to be inconsistent (i.e., unsatisfiable), the solver will return `UNSAT` cores, which are then communicated to the user to pinpoint the precise requirements causing the inconsistency which can aid users to effectively address the identified issues. Following the successful completion of consistency checks, the guarantee statements will undergo validity analysis. The guarantee analysis involves checking the satisfiability of $\neg(\alpha_{r1} \wedge ... \wedge \alpha_{rn} \implies \beta_R)$ in SMT. Note that the SMT formula is the negation ($\neg$) of the implication because SMT solvers prove the validity of logical statements by contradiction. An `UNSAT` response from SMT solvers indicates that the left-hand side (LHS) requirements of the guarantee do imply the right-hand side (RHS) requirement; whereas, a `SAT` answer signifies the opposite. When the guarantee statement is found to be invalid (i.e., a `SAT` answer), the SMT solver will provide a counterexample. It consists of a set of variable assignments that satisfies all of the input formulae and demonstrates why the guarantee is invalid. This counterexample information will also be reported back to the users, and will help them to debug and refine the requirements, ensuring that they accurately capture the intended constraints.

Manually entering guarantee statements can be tedious. To streamline this process, we developed a feature that automatically traces dependent requirements for the consequent of guarantees. This traceability analysis traverses the entire requirements set and traces the transitive dependencies of the variables within the consequent requirement. Eventually, it gathers all associated requirements that influence variables of the consequent requirement. However, it is important to note that the included dependent requirements may encompass irrelevant ones that do not impact the validity of the guarantees. Users have the flexibility to select which requirements to include in the guarantee validity analysis. Filtering irrelevant requirements is planned as future work.

5 Evaluation

Hybrid Electric Propulsion System (HEPS) Use Case. In recent decades, electrified aircraft propulsion (EAP) has emerged as a vital solution in reducing carbon dioxide emissions and improving aircraft efficiency. Among the various EAP systems, parallel hybrid electric propulsion system (HEPS) makes it feasible to target national/international aircraft markets with distance ranges greater than 3000 mi by maintaining optimum weight and providing fuel reductions [23,37]. Without incorporating an electric generator, parallel HEPS has the advantage of higher overall efficiency at cruise speeds. Figure 1 shows a parallel HEPS architecture model in the model-based system engineering environment called Cameo Systems Modeler, wherein the traditional gas turbine engine power is supplemented by a battery-powered electric motor to provide the required power. Cameo Systems Modeler allows for the captures of system architecture and requirements and is based upon the UML2 modeling language. Figure 1 shows various systems and subsystems of an aircraft within a parallel HEPS depicted in a hierarchical structure. Objects marked by «» denote that they belong to their assigned *stereotypes*, which is a UML2 modeling concept.

In addition to UML2, Cameo System Modeler uses SysML for systems modeling and is an extension of UML2. In particular, we use the SysML sterotypes: *system, subsystem* and *block* for parallel HEPS Cameo model. More details about Cameo and SysML can be found elsewhere [12]. To perform requirements analysis, the system architecture description and requirements captured in Cameo are manually formalized in the TRACE IDE. The architecture is used as guidance to declare the component types and their instances. Only a subset of requirements amenable to TRACE analyses are formalized in the TRACE IDE. In this configuration, both the turbine and the motor can feed power to the propeller fan separately or together. The power sharing between them is regulated, targeting improved overall energy efficiency of the system. In order to evaluate and validate TRACE, a parallel HEPS (shown in Fig. 1) is taken up as a case study. HEPS is well-suited for TRACE, as its requirement specification consists of traditional functional requirements in addition to multi-domain requirments involving power electronics and thermal domains. An open-source version of TRACE HEPS model is available[1].

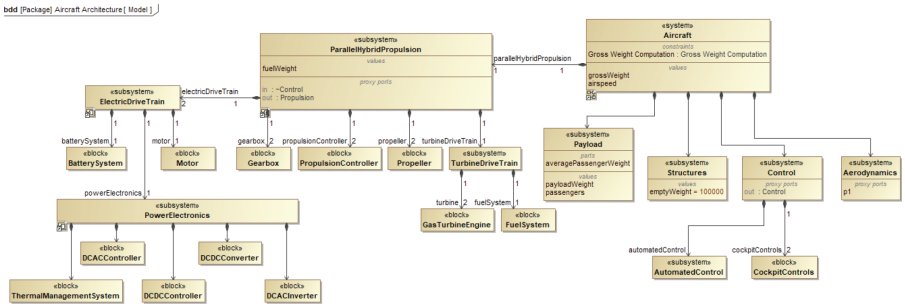


Fig. 1. Hybrid-Electric Propulsion Architecture in Cameo

HEPS Requirements Analysis. The architecture and textual requirements are first modeled in Cameo. Subsequently, each system and controller specified in the Cameo model are manually formalized in TRACE. HEPS consists of several subsystems including Gas Turbine, Power Converters (providing electrical power) and system controllers which toggle logical states upon specified monitored variable conditions. For example, the `DCDCConverter` is a subsystem that performs DC to DC power conversion from a Battery system power source. A `DCACConverter` subsequently converts the DC power output from `DCDCConverter` into AC power to be delivered to the engine. These power conversion systems are augmented by their respective power conversion controllers.

[1] https://github.com/ge-high-assurance/HEPS.

```
Units { V : "volts",A : "ampere", ohm : "ohm", rpm : "rpms",celsius : "celsius",ft : "feet",
        kwh : "kilo-watt hours",  kgs:"kilograms",Wh_per_kg:"Watt-hour per kilogram" }
Mode is an enum with values
 [ After_Landing_Roll, Descent, Final_Roll, Final_Approach, GroundIdle].
```

```
Switch is an enum type with values [ON, OFF].
PWMState is an enum type with values [Enable, Disable].
Controller is a component type with variables{
    pwm : PWMState
    fault : real
    temp : real with unit celsius
    voltage : real with unit V }
DCDCController is a type of Controller.
DCACController is a type of Controller with owned variables { dc_voltage : real with unit V }
Aircraft is a component type with variables {mode : Mode, EGT_light_exceedance : Switch }
Motor is a component type with variables { motor_speed : real with unit rpm }
max_speed : real with value 2000 rpm.
```

One critical requirement of the power conversion controllers is to signal a fault state when the power converter temperatures exceed allowed thresholds. The architecture of HEPS system with its constituent (sub-)sytems and (sub-)system controllers are shown in Fig. 1. For the sake of illustration, we focus our attention on a set of requirements involving DC-DC controller and DC-AC controller. We focus on other high-level requirements later for inter-domain interaction of requirements.

Conflict Analysis. An engineer can specify the behavior of `DCACController` using the following two requirements.

```
Requirement DCACControllerR1:
   DCACController shall set dcac_controller_pwm to Enable
   when (Aircraft:mode = Takeoff and dcac_controller_fault = 0.0)
   or (Aircraft:mode = Climb and dcac_controller_fault = 0.0).
Requirement DCACControllerR2:
   DCACController shall set dcac_controller_pwm to Disable
   when Aircraft:mode = Taxi or Aircraft:mode = Cruise
   or Aircraft:mode = Descent or Aircraft:mode = Final_Approach
   or Aircraft:mode = After_Landing_Roll or dcac_controller_fault = 0.0.
```

Requirements `DCACControllerR1` and `DCACControllerR2` are flagged as conflicting by TRACE because when combining both the requirements, the DCAC-Controller is setting `PWM` (Pulse-Width Modulation) to `Enable` in the first requirement and setting `PWM` to `Disable` in the second requirement for the same monitored variable conditions. Upon changing `dcac_controller_fault to 1.0` in `DCACControllerR2`, we can resolve the conflict. Other conflicting requirements are shown in Table 3.

Independence and Contingency Analysis. Independence analysis reveals that certain requirements are inherently implied by others. In the HEPS requirements, multiple instances of such dependencies are shown in Table 3, potentially stemming from oversights in engineers' encoding practices, inconsistent communications, or ambiguous documentation. Contingent requirements are those that can be both feasibly fulfilled and proven false during execution.

Table 3. Requirement analyses report snippets in TRACE IDE

Requirement conflict analysis results snippet	
Requirement	**Conflicting Requirement**
CockpitControl:CockpitControl_CCC13	CockpitControl:CockpitControl_CCC11
DCDCCtrReqs:DCDCCtrR1	DCDCCtrReqs:DCDCCtrR2
Requirements independent analysis results snippet	
Requirement	**Dependent Requirement**
CockpitControl:CockpitControl_CCC11	CockpitControl:CockpitControl_CCC5
CockpitControl:CockpitControl_CCC3	CockpitControl:CockpitControl_CCC5
DCDCCtrReqs:DCDCCtrR1	DCDCCtrReqs:DCDCCtrR2
Requirements contingency analysis results snippet	
Unsatisfiable Requirement	**Tautology Requirement**
CACCtrReqs:DCACCtrR3a	CockpitControl:CockpitControl_CCC5
Requirements global contingency analysis results snippet	
Globally Unsatisfiable Requirement Package	**Tautology Requirement Package**
HLRs	

```
StructuresEnum is an enum type with values [Deploy].
Requirement CockpitControl_CCC5:
  Pilot shall set Structures:landing_gear to Deploy when Aircraft:mode = Final_Approach.
max_speed : real with value 5100 rpm.
Requirement DCACCtrR3a:
  DCACController shall set max_speed to 1.0 when Motor:max_speed > max_speed.
```

A common issue with unsatisfiable requirements could be contradictory encoding. Requirements that are infallible would be considered tautology, i.e., is true by definition. They do not provide any meaningful or actionable information beyond what is already known. For example, requirement `CockpitControl_CCC5` sets `Structures:landing_gear` to an enumerate type `StructuresEnum`, which only has one value `Deploy`. Therefore, the requirement will always be true regardless of the *when* condition. TRACE can identify contingent requirements as well as unsatisfiable and infallible requirements as shown in Table 3. Finally, we also check the contingency of the conjunction of all requirements in a package. The per-package result report is shown in Table 3.

Completeness Analysis. Completeness analysis can reveal issues in underspecified requirements, which are often caused by evolving project scope, inadequate analysis of user needs or systems. As shown in Table 4, the 2 requirements in package `AutomatedControl` are incomplete.

Table 4. Requirements completeness analysis results snippet.

Package Name	Incomplete Requirements	Complete Requirements
AutomatedControl	AutomatedControl_CAC5; AutomatedControl_CAC5a	
DCACCtrReqs		CDACCtrR1; DCACCtrR2; DCACCtrR3; DCACCtrR4; DCACCtrR5; DCACCtrR6; DCACCtrR7; DCACCtrR8;

```
Requirement AutomatedControl_CAC5:
  Auto_control shall set  Aircraft:aircraft_EGT_light_exceedance to ON
  when GasTurbine:gas_turbine_EGT > EGT_permissible.
Requirement AutomatedControl_CAC5a:
  Auto_control shall set  Aircraft:aircraft_EGT_light_exceedance to OFF
  when GasTurbine:gas_turbine_EGT < EGT_permissible.
```

Further analysis determines a gap in the requirements, specifically regarding the condition where `GasTurbine:gas_turbine` equals `EGT_permissible`. One way to address the issue is to update the when condition of `AutomatedControl_CAC5a` to `GasTurbine:gas_turbine_EGT <= EGT_permissible`, encompassing the equality scenario. A total of 79 requirements were checked for conflicts, contingency, completeness, independence and global contingency within an average of 1.673 s running on a 6-core Intel(R) Core(TM) i7-10850H CPU @ 2.70 GHz with 32 GB RAM on Windows 10 platform.

Multi-domain Requirements Analysis in TRACE. We describe how requirements from `BatteryDomain`, `PowerElectronicsDomain`, & `Thermal Domain` are analyzed in a guarantee statement. In HEPS, the Battery provides usable power to DCDCConverter, whose output power is inverted into AC using a DCACConverter. The inverted AC power is fed to the motor that drives the engine under appropriate aircraft modes of operation. While performing power conversion from DC to DC and DC to AC, the power electronics within the converters generate heat that needs to be rejected to a Chiller.

```
    Outlet_T_max : real with value 100 celsius.
    Battery is a component type with variables {
        usable_storage_power : real with unit kW as a property
        weight : real with unit kgs as a property }
    PowerConverter is a component type with variables {
        loss : real with unit kW as a property
        efficiency : real as a property
        temperature : real with unit celsius as a property
        weight : real with unit kgs
        resistance : real with unit ohm as a property }
    DCDCConverter is a type of PowerConverter.
    DCACConverter is a type of PowerConverter.
    Chiller is a component type with variables{
        inlet_temperature : real with unit celsius as a property
        outlet_temperature: real with unit celsius as a property }
    BatteryDomain domain batteryDomain {
        Requirement BatteryPower : Battery shall provide usable_storage_power = 100 kW.  }
```

If the temperature at the junctions of both converters exceed a certain threshold, the PowerController will set a fault state to signal a faulty condition. Even at requirements phase, the maximum ranges of junction temperatures can be captured and checked against the tolerance thresholds as guarantees. Both DCDC-Converter & DCAC-Converter are subtypes of Converter. The system description is provided: note that the variables are physical properties of the power converter systems as opposed to variables in the power controller that measures the state of the system using sensors. Note that `Outlet_T_max` is a global constant that constraints the Chiller outlet temperature. The `PowerElectronicsDomain` specifies the power conversion efficiency requirements with the battery input source along with equations for the energy losses incurred during power conversion.

```
PowerElectronicsDomain domain powerElectronicsDomain {
  Requirement DCDC_Efficiency: DCDCConverter shall provide efficiency >= 0.96 .
  Requirement DCDC_Efficiency_lt_1: DCDCConverter shall provide efficiency < 1 .
  Requirement DCAC_Efficiency: DCACConverter shall provide efficiency >= 0.99.
  Requirement DCAC_Efficiency_lt_1: DCACConverter shall provide efficiency < 1.
  Equation DCDC_Loss: DCDCConverter::loss=(1-efficiency) * Battery:usable_storage_power kW.
  Equation DCAC_Loss: DCACConverter::loss=(1-efficiency) * DCDCConverter:loss kW. }
```

The `ThermalDomain` requirements specify equations for the power converter(s) junction temperature calculation along with constraints on the Chiller outlet temperature ranges. Requirements `DCAC_Junction_Temperature` & `DCDC_Junction_Temperature` should be checked if the equations and requirements involved in the junction temperature calculations satisfy the specified temperature ranges.

```
ThermalDomain domain thermalDomain{
 Equation DCDC_Junction_Temperature:
  DCDCConverter::temperature= (Chiller:outlet_temperature + loss * resistance ) celsius.
 Equation DCAC_Junction_Temperature:
  DCACConverter::temperature= (Chiller:outlet_temperature + loss * resistance) celsius.}
Requirement DCDC_Junction_Temp_Range:
DCDCConverter shall provide temperature<=175 celsius.
Requirement DCAC_Junction_Temp_Range:
DCACConverter shall provide temperature<=175 celsius.
Requirement Chiller_Outlet_Temp:
Chiller shall provide outlet_temperature<=(Outlet_T_max) celsius.
Requirement DCDC_Thermal_Resistance: DCDCConverter shall provide resistance=1 C_per_kW.
Requirement DCAC_Thermal_Resistance: DCACConverter shall provide resistance=1 C_per_kW.  }
```

Guarantee Validity Analysis

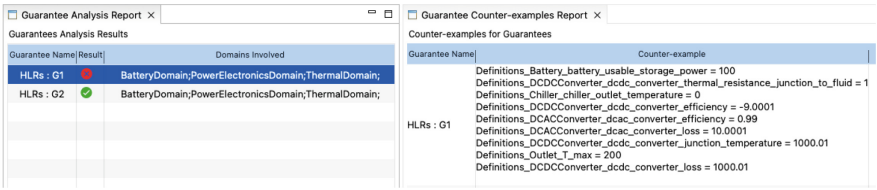
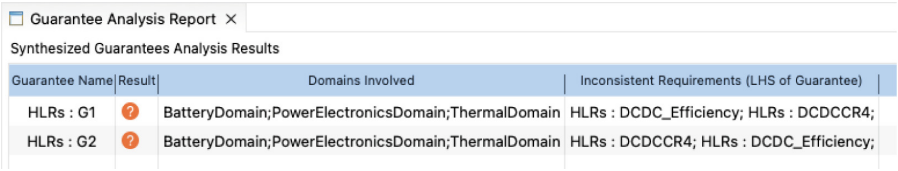


Fig. 2. Guarantee analysis multi-domain requirements & counter-example for G1

All the multi-domain requirements that may potentially be involved in the calculation of junction temperature ranges are assembled into a guarantee statement:

```
Guarantee G1 "DCDC Junction Temperature below a Threshold":
BatterySystem, DCDC_Loss, DCDC_Efficiency,  DCDC_Efficiency_lt_1, Chiller_Outlet_Temp,
DCDC_Junction_Temperature, DCDC_Thermal_Resistance imply DCDC_Junction_Temp_Range.
Guarantee G2 "DCAC Junction Temperature below a Threshold":
BatterySystem, DCDC_Loss, DCDC_Efficiency, DCAC_Efficiency, DCDC_Junction_Temperature
DCAC_Junction_Temperature, DCAC_Thermal_Resistance, Chiller_Outlet_Temp, DCAC_Loss,
DCDC_Thermal_Resistance, DCDC_Junction_Temp_Range imply DCAC_Junction_Temp_Range.
```

Note that the antecedent for guarantee G2 must be stronger as the DC-AC power conversion relies on the DC-DC power conversion sub-system. Given these inter-domain requirements and guarantees, TRACE IDE will check the guarantees for validity using the semantics by discharging an equivalent SMT query. However, if the requirements author misses an equation, either of the guarantees are falsifiable and fail the TRACE IDE analysis. TRACE assembles a report about the status of the guarantees along with the different domains involved in the guarantee statements. Figure 2 shows the guarantee report produced by TRACE IDE, where guarantee G1 is invalid and G2 is valid.


Guarantee Analysis Report ×

Synthesized Guarantees Analysis Results

Guarantee Name	Result	Domains Involved	Inconsistent Requirements (LHS of Guarantee)
HLRs : G1	?	BatteryDomain;PowerElectronicsDomain;ThermalDomain	HLRs : DCDC_Efficiency; HLRs : DCDCCR4;
HLRs : G2	?	BatteryDomain;PowerElectronicsDomain;ThermalDomain	HLRs : DCDCCR4; HLRs : DCDC_Efficiency;

Fig. 3. Guarantee's antecedent consistency analysis.

Prior to the guarantee checks, we ensure the consistency of the antecedent by verifying it is not trivially false. This check precludes guarantees that hold true vacuously due to inconsistent antecedents. For example, an engineer might erroneously draft a requirement for DCDCConverter efficiency to be less than 96% (instead of greater than 96%) in a requirement `DCDCCR4`. This would make both guarantees trivially valid as the antecedent is false.

```
Requirement DCDCCR4: DCDCConverter shall provide efficiency < 0.96.
```

We precisely identify and report back to users the set of requirements that are inconsistent. Behind the scenes, the `UNSAT` core is processed to report inconsistent requirements. An example demonstrating this feature is shown in Fig. 3, with highlighted requirements `DCDC_Efficiency` and `DCDCCR4` in package `HLRs` shown as inconsistent. When the analysis fails, counterexamples will be generated for failed guarantees. A counter-example for guarantee `G1` is shown in Fig. 2.

Guarantee Traceability Analysis. We invoke the TRACE traceability function on `G1` and `G2` to collect and reports their dependent requirements, see Fig. 4.

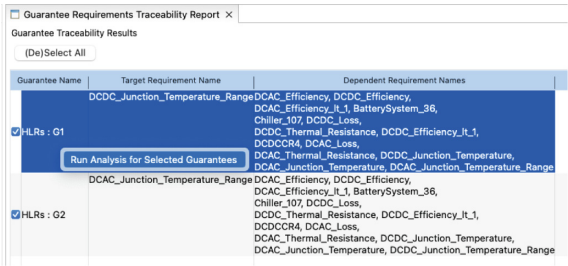


Fig. 4. Guarantee traceability analysis report.

6 Related Work

Several tools have been proposed in the literature for capturing and analyzing requirements [29]. ASSERT [10,28] takes controlled English language requirements and can perform contingency, independence, conflict, and completeness analysis using the ACL2 [27] theorem prover. While ASSERT has several features of TRACE such as type and unit checking, TRACE uses an SMT backend and can capture and analyze multi-domain sytems and low-level functional requirements. FRET [26] can be used for writing and formalizing temporal requirements and perform correctness and realizability analysis using a collection of back-end formal tools. While FRET provides rich support to model requirements that require state-transition abstraction for timed requirements, it lacks capture of units and a hierarchical system description like TRACE, which are crucial to large scale systems engineering projects. SpeAR [14,20,36] supports consistency and completeness checking using a back-end model checker. Prospec 2.0 [18] allows users to develop formal requirements and validate them using linear temporal logic (LTL). STIMULUS [24] supports early debugging and validation of functional real-time requirements written in a textual format. The EARS-CTRL tool [30] can be used for correctness and realizability testing. RATSY [4] uses automata to analyze for correctness, realizability, and completeness. SPECTRA [31] uses reactive systems language to formalize requirements and can perform vacuity and well-separation analysis in addition to consistency and realizability. AGREE [17] and Pacti [22] frameworks are used to analyze assume-guarantee requirements for consistency and realizability. ARSENAL [13,19] can be used to formalize and analyze natural language requirements for realizability and consistency. CHASE [34] and CLEAR [3] can both be used for analysing correctness, completeness, consistency, and realizability. Table 5 compares TRACE to the important properties of several requirement analyses languages and tools.

Table 5. Comparison of TRACE with requirements analyses languages and tools

Language	System Reqs	Multi-domain	Low-level Reqs	Units	Types	LTL	SMT	Pseudo NL	Analyses	Reasoner	Case Study
FRET	×	×	✓	×	✓	✓	✓	✓	Conflict, Consistency, Realizability	JKind [16], Kind2 [7], NuSMV [9]	✓
SpeAR	✓	×	✓	✓	✓	✓	×	✓	Consistency, Completeness	JKind	✓
Prospec 2.0	×	×	✓	×	✓	✓	✓	✓	Satisfiability	SPIN [21], NuSMV	✓
STIMULUS	×	×	✓	×	✓	×	×	✓	Simulation	Simulink [32]	✓
EARS-CTRL	×	×	✓	×	✓	✓	×	✓	Correctness, Realizability	autoCode4 [8]	✓
RATSY	×	×	✓	×	×	✓	×	×	Consistency, Realizability, Correctness	NuSMV, CUDD [35], Anzu [25]	˜ ✓
ARSENAL	×	×	✓	×	✓	✓	×	✓	Consistency, Realizability	SAL [11]	✓
CHASE	✓	×	✓	×	✓	✓	×	✓	Consistency, Assume Guarantee, Realizability	TuLiP [15]	✓
CLEAR	×	×	✓	✓	✓	✓	✓	✓	Consistency, Completeness	NuSMV, DiVinE [2], Acacia [5], z3 [33]	✓
AGREE	×	×	✓	×	✓	✓	✓	×	Consistency, Realizability	JKind, z3	✓
Pacti	✓	×	✓	×	✓	×	×	×	Planning, Constraint-solving	Linear Programming Algorithms	✓
ASSERT	×	×	✓	✓	✓	×	×	✓	Consistency, Conflict, Contingency, Mode analyses	ACL2s [6]	✓
TRACE	✓	✓	✓	✓	✓	×	✓	✓	Consistency, Conflict, Contingency, Guarantee, Multi-domain	z3, cvc5 [1]	✓

7 Conclusion and Future Work

In this paper, we have presented a language and tool – TRACE for capturing and analyzing formal system requirements. Our contributions include the introduction of a new language tailored for expressing multi-domain system requirements, along with a robust toolset for syntax validation, semantic analysis, and unit checking. Furthermore, we provide a formal description of both the language syntax and semantics with illustrative examples. Specifically, we conducted a comprehensive evaluation of TRACE on an industrial HEPS, which highlighted its practical utility and effectiveness in addressing complex system

requirements. In the context of model-based systems engineering, we anticipate a seamless integration of the formalization process of Cameo requirements into TRACE and their analyses results reported directly in Cameo.

In future work, we plan to enhance TRACE by incorporating support for temporal requirements capture with backend model checkers for analysis. Our roadmap also includes the integration of optimization modulo theories (OMT) solvers into TRACE. This extension will empower the tool to tackle complex optimization problems, thereby broadening its applicability across domains where optimization challenges are prevalent.

Acknowledgement & Disclaimer. Distribution Statement "A" (Approved for Public Release, Distribution Unlimited). This research was developed with funding from the Defense Advanced Research Projects Agency (DARPA). The views, opinions and/or findings expressed are those of the author and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

References

1. Barbosa, H., et al.: cvc5: a versatile and industrial-strength SMT solver. In: Fisman, D., Rosu, G. (eds.) TACAS 2022. LNCS, vol. 13243, pp. 415–442. Springer (2022). https://doi.org/10.1007/978-3-030-99524-9_24
2. Barnat, J., Brim, L., Češka, M., Ročkai, P.: DiVinE: parallel distributed model checker. In: 2010 Ninth International Workshop on Parallel and Distributed Methods in Verification, and Second International Workshop on High Performance Computational Systems Biology, pp. 4–7. IEEE (2010). https://doi.org/10.1109/PDMC-HiBi.2010.9
3. Bhatt, D., Murugesan, A., Hall, B., Ren, H., Jeppu, Y.: The CLEAR way to transparent formal methods. In: 4th Workshop on Formal Integrated Development Environment (2018)
4. Bloem, R., et al.: RATSY – a new requirements analysis tool with synthesis. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 425–429. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14295-6_37
5. Bohy, A., Bruyère, V., Filiot, E., Jin, N., Raskin, J.-F.: Acacia+, a tool for LTL synthesis. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 652–657. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31424-7_45
6. Chamarthi, H.R., Dillinger, P., Manolios, P., Vroon, D.: The ACL2 sedan theorem proving system. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 291–295. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-19835-9_27
7. Champion, A., Mebsout, A., Sticksel, C., Tinelli, C.: The Kind 2 model checker. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016, Part II. LNCS, vol. 9780, pp. 510–517. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41540-6_29
8. Cheng, C.-H., Lee, E.A., Ruess, H.: autoCode4: structural controller synthesis. In: Legay, A., Margaria, T. (eds.) TACAS 2017, Part I. LNCS, vol. 10205, pp. 398–404. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54577-5_23

9. Cimatti, A., et al.: NuSMV 2: an OpenSource tool for symbolic model checking. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 359–364. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45657-0_29
10. Crapo, A., Moitra, A., McMillan, C., Russell, D.: Requirements capture and analysis in ASSERT™. In: 2017 IEEE 25th International Requirements Engineering Conference (RE), pp. 283–291. IEEE (2017). https://doi.org/10.1109/RE.2017.54
11. de Moura, L., et al.: SAL 2. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 496–500. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-27813-9_45
12. Delligatti, L.: SysML Distilled: A Brief Guide to the Systems Modeling Language. Addison-Wesley (2013). https://doi.org/10.5555/2560076
13. Elenius, D., Yeh, E., Graham-Lengrand, S., Ghosh, S., Lincoln, P., Shankar, N.: Deriving formal specifications from natural language requirements using arsenal 2. In: High Confidence Software and Systems Conference (2019)
14. Fifarek, A.W., Wagner, L.G., Hoffman, J.A., Rodes, B.D., Aiello, M.A., Davis, J.A.: SpeAR v2.0: formalized past LTL specification and analysis of requirements. In: Barrett, C., Davies, M., Kahsai, T. (eds.) NFM 2017. LNCS, vol. 10227, pp. 420–426. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-57288-8_30
15. Filippidis, I., Dathathri, S., Livingston, S.C., Ozay, N., Murray, R.M.: Control design for hybrid systems with TuLiP: the temporal logic planning toolbox. In: 2016 IEEE Conference on Control Applications (CCA), pp. 1030–1041. IEEE (2016). https://doi.org/10.1109/CCA.2016.7587949
16. Gacek, A., Backes, J., Whalen, M., Wagner, L., Ghassabani, E.: The JKind model checker. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10982, pp. 20–27. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96142-2_3
17. Gacek, A., Katis, A., Whalen, M.W., Backes, J., Cofer, D.: Towards realizability checking of contracts using theories. In: Havelund, K., Holzmann, G., Joshi, R. (eds.) NFM 2015. LNCS, vol. 9058, pp. 173–187. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-17524-9_13
18. Gallegos, I., Ochoa, O., Gates, A.Q., Roach, S., Salamah, S., Vela, C.: A property specification tool for generating formal specifications: Prospec 2.0. In: SEKE, pp. 273–278 (2008)
19. Ghosh, S., Elenius, D., Li, W., Lincoln, P., Shankar, N., Steiner, W.: ARSENAL: automatic requirements specification extraction from natural language. In: Rayadurgam, S., Tkachuk, O. (eds.) NFM 2016. LNCS, vol. 9690, pp. 41–46. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-40648-0_4
20. Gross, K.H., Fifarek, A.W., Hoffman, J.A.: Incremental formal methods based design approach demonstrated on a coupled tanks control system. In: 2016 IEEE 17th International Symposium on High Assurance Systems Engineering (HASE), pp. 181–188. IEEE (2016). https://doi.org/10.1109/HASE.2016.16
21. Holzmann, G.J.: The model checker spin. IEEE Trans. Software Eng. **23**(5), 279–295 (1997). https://doi.org/10.1109/32.588521
22. Incer, I., Badithela, A., Graebener, J.B., Mallozzi, P., Pandey, A., Rouquette, N., Yu, S.J., Benveniste, A., Caillaud, B., Murray, R.M., et al.: Pacti: assume-guarantee contracts for efficient compositional analysis and design. ACM Trans. Cyber-Phys. Syst. (2024). https://doi.org/10.1145/3704736
23. Jansen, R.H., Bowman, C.L., Clarke, S., Avanesian, D., Dempsey, P.J., Dyson, R.W.: NASA electrified aircraft propulsion efforts. Aircr. Eng. Aerosp. Technol. **92**(5), 667–673 (2020). https://doi.org/10.1108/AEAT-05-2019-0098

24. Jeannet, B., Gaucher, F.: Debugging embedded systems requirements with stimulus: an automotive case-study. In: 8th European Congress on Embedded Real Time Software and Systems (ERTS 2016) (2016)
25. Jobstmann, B., Galler, S., Weiglhofer, M., Bloem, R.: Anzu: a tool for property synthesis. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 258–262. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73368-3_29
26. Katis, A., Mavridou, A., Giannakopoulou, D., Pressburger, T., Schumann, J.: Capture, analyze, diagnose: realizability checking of requirements in FRET. In: Shoham, S., Vizel, Y. (eds.) CAV 2022, Part II. LNCS, vol. 13372, pp. 490–504. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-13188-2_24
27. Kaufmann, M., Moore, J.S.: ACL2: An industrial strength version of Nqthm. In: Proceedings of 11th Annual Conference on Computer Assurance, COMPASS 1996, pp. 23–34. IEEE (1996). https://doi.org/10.1109/CMPASS.1996.507872
28. Li, M., et al.: Requirements-based automated test generation for safety critical software. In: 2019 IEEE/AIAA 38th Digital Avionics Systems Conference (DASC), pp. 1–10. IEEE (2019). https://doi.org/10.1109/DASC43569.2019.9081726
29. Lorch, R., et al.: Formal methods in requirements engineering: survey and future directions. In: International Conference on Formal Methods in Software Engineering (FormaliSE 2024) (2024). https://doi.org/10.1145/3644033.3644373
30. Lúcio, L., Rahman, S., Cheng, C.-H., Mavin, A.: Just formal enough? Automated analysis of EARS requirements. In: Barrett, C., Davies, M., Kahsai, T. (eds.) NFM 2017. LNCS, vol. 10227, pp. 427–434. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-57288-8_31
31. Maoz, S., Ringert, J.O.: Spectra: a specification language for reactive systems. Softw. Syst. Model. **20**(5), 1553–1586 (2021). https://doi.org/10.1007/s10270-021-00868-z
32. MathWorks, Inc.: Mathworks Simulink. https://www.mathworks.com/products/simulink.html
33. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
34. Nuzzo, P., Lora, M., Feldman, Y.A., Sangiovanni-Vincentelli, A.L.: CHASE: contract-based requirement engineering for cyber-physical system design. In: 2018 Design, Automation & Test in Europe Conference & Exhibition (DATE), pp. 839–844. IEEE (2018). https://doi.org/10.23919/DATE.2018.8342122
35. Somenzi, F.: CUDD: CU decision diagram package-release 2.4. 0. Univ. Colorado Boulder **21** (2009)
36. Wagner, L.: Formal specification and analysis of requirements using SpeAR. ACM SIGAda Ada Lett. **39**(1), 20–34 (2020). https://doi.org/10.1145/3379106.3379110
37. Wheeler, P., Sirimanna, T.S., Bozhko, S., Haran, K.S.: Electric/hybrid-electric aircraft propulsion systems. Proc. IEEE **109**(6), 1115–1127 (2021). https://doi.org/10.1109/JPROC.2021.3073291