Contract-Based Integration of Cyber-Physical Analyses

Ivan Ruchkin
Inst. for Software Research
Carnegie Mellon University
Pittsburgh, PA, USA
iruchkin@cs.cmu.edu

Dionisio De Niz
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA, USA
dionisio@sei.cmu.edu

Sagar Chaki
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA, USA
chaki@sei.cmu.edu

David Garlan
Inst. for Software Research
Carnegie Mellon University
Pittsburgh, PA, USA
garlan@cs.cmu.edu

ABSTRACT

Developing cyber-physical systems involves multiple engineering domains, e.g., timing, logical correctness, thermal resilience, and mechanical stress. In today's industrial practice, these domains rely on multiple analyses to obtain and verify critical system properties. Domain differences make the analyses abstract away interactions among themselves, potentially invalidating the results. Specifically, one challenge is to ensure that an analysis is never applied to a model that violates the assumptions of the analysis. Since such violation can originate from the updating of the model by another analysis, analyses must be executed in the correct order. Another challenge is to apply diverse analyses soundly and scalably over models of realistic complexity. To address these challenges, we develop an analysis integration approach that uses *contracts* to specify dependencies between analyses, determine their correct orders of application, and specify and verify applicability conditions in multiple domains. We implement our approach and demonstrate its effectiveness, scalability, and extensibility through a verification case study for thread and battery cell scheduling.

General Terms

Verification, Design, Theory

Keywords

Cyber-physical systems, analysis, real-time scheduling, thermal runaway, model checking, battery scheduling, analysis contracts, virtual integration

1. INTRODUCTION

The development of today's industrial-scale cyber-physical systems (CPS) is heavily driven by models [15] and analyses. This trend is expected to continue, since it enables CPSs to be developed, upgraded, integrated, and verified virtually through models before manufacturing starts. Modeling also supports collaborative development by different teams, and fosters early error detection, faster development, and lower costs. In particular, analyses enable model creation and verification at design time to guarantee important quality attributes, such as control stability, schedulability, power consumption, safety, and security.

These analyses emerge from different engineering domains, such as timing, logical correctness, and thermal resilience. Consequently, they focus on different CPS abstractions that interact in subtle ways. This leads to two problems that today render analysis results untrustworthy: (i) one analysis modifies a system model in a way that violates the assumptions made by another. E.g., a real-time task-allocation algorithm [8] assigns a set of threads to a processor scheduled via a dynamic priority algorithm, thus violating the fixed priority assumption made by a model checker [4]. Also, (ii) the specification of such assumptions and the detection of their violation are left implicit in the hands of human designers that, more often than not, are unable to cope with their complexity and subtlety due to the inter-domain nature of the assumptions.

As a consequence, these problems are currently discovered late during system integration, leading to costly fixes [25]. This issue is particularly prevalent in industries with complex supply chains, such as avionics and automotive, where systems are integrated from independently-developed parts, designs of which are analyzed with a mishmash of tools. Complexity of integration is currently mitigated in an ad-hoc and manual way, which is neither scalable, nor able to provide a high degree of assurance [14].

In this paper, we present and evaluate an alternative two-part solution that is mathematically rigorous and automated. The first part is an analysis *contract* specification language with well-defined syntax and semantics. A contract for an analysis expresses both the *assumption* under which it produces a sound result, and the *guarantee* about the resulting modified model. Our contract language finds a balance between expressiveness and decidability to support contracts that capture both design-time and runtime system aspects. An example is the frequency scaling analysis [23], which assumes that the runtime scheduling of threads allocated at

ESWEEK '14, October 12-17, 2014, New Delhi, India

http://dx.doi.org/10.1145/2656045.2656052

design-time to the same CPU is behaviorally equivalent to deadline monotonic. To this end, our contract language combines a many-sorted first-order logic with a variant of linear temporal logic, propositions of which are derived from predicates over the system state. The logical nature of the language also makes it applicable to multiple analysis domains. In addition, validity of contracts expressed in our language is efficiently decidable.

The second part of our solution is a contract verification algorithm that ensures soundness of analysis results. The algorithm takes a set of analysis contracts and an architectural CPS model as input, computes the dependencies between the contracts, and executes the analyses over the model in an order guaranteed to produce sound results. During this process, our algorithm checks the validity of each contract: (i) before an analysis is executed, the validity of its assumption is checked over the input model; and (ii) after the analysis completes, the validity of its guarantee is ensured over the output model. Moreover, the validity is checked in a sound and exhaustive manner via co-operative application of an SMT solver (for the design-time aspect) and a model checker (for the runtime aspect) of the contract.

Finally, we implement our approach and demonstrate its effectiveness, scalability, and extensibility with a case study that involves multiple analyses from two domains: real-time thread scheduling and battery cell scheduling. Our implementation is based on OSATE [20], which enables us to handle CPS models described in the Architecture Analysis and Design Language (AADL) [9], an SAE standard. It also uses state-of-the-art tools Z3 [7] and Spin [13] for SMT solving and model checking, respectively. We show how our approach detects subtle bugs in interactions between analyses, and how the use of advanced tools enables it to scale to systems of realistic complexity.

The rest of the paper is organized as follows. Sec. 2 introduces a running example of a system and analyses used in its design. Sec. 3 presents the analysis contract language. Sec. 4 presents our contract verification algorithm. Sec. 5 and Sec. 6 present our implementation and evaluation, respectively. We wrap up by surveying related work in Sec. 7 and conclude in Sec. 8.

2. RUNNING EXAMPLE

Consider a reconnaissance aircraft as an example CPS. It is controlled by a set of threads (a.k.a. tasks) with different security levels executing on several processors (a.k.a. CPUs). Each thread executes an infinite sequence of periodic jobs. A job is a finite computation, e.g., a control correction for aircraft stability. The system has dynamic multi-cell batteries with configurable connections between cells so that some cells recharge while others are discharging [12]. Thread scheduling policies, CPU frequencies, battery cell scheduling policies, and allocation of threads to CPUs are among adjustable parameters in the system's design.

The system has to satisfy several requirements: (i) data security – threads with different security levels should not run on the same CPU; (ii) schedulability – all jobs must meet deadlines required by the control algorithms; (iii) energy efficiency – CPUs must operate on the minimal frequency possible, thus maximizing battery life; (iv) safe concurrency – threads must be free of deadlocks and race conditions; (v) thermal safety – even if a battery cell overheats, it should not trigger a chain reaction called *thermal runaway* [5].

All of these requirements can be verified individually with analysis tools from different engineering domains. However, ensuring that all the requirements are satisfied together is challenging, since one tool can select some design parameters that violate the assumptions of another, as discussed before. Similarly, creating a single overarching analysis that takes into account all requirements is intractable. Our approach uses each analysis as is, but complements them with analysis contracts that allow us to model their interdependencies and ensure their sound application. This enables sound and tractable verification of all the system requirements.

As an illustration, we consider the following set of analyses $\mathcal{AN}$ through the rest of the paper: (i) secure thread allocation: computes permissible thread co-locations based on security levels; (ii) bin packing [8]: assigns threads on CPUs to ensure schedulability; (iii) frequency scaling: minimizes the CPU frequency given the threads assignment; (iv) REK model checking [4]: checks if threads satisfy user-specified safety properties like absence of race conditions and deadlocks; (v) thermal runaway: determines patterns of battery cell connections that lead to thermal runaway; and (vi) battery scheduling: determines a battery scheduler given the required operation time and battery size.

Arbitrary independent use of these analyses can lead to unsound results. For example, running bin packing before thread allocation could violate secure co-location constraints set by the latter, but not present at the time we run the former. Similarly, using the frequency scaling algorithm that assumes a deadline-monotonic scheduler on a CPU that uses the earliest deadline first scheduler would produce unsound results. In this paper we demonstrate how our approach enables systematic integration of these analyses ensuring satisfaction of their interdependencies and assumptions.

3. CONTRACT SPECIFICATION

In this section we present our contract language, and use it to formalize contracts for the analyses from Sec. 2.

3.1 Analysis Verification Domain

First, we formalize concepts on which engineering domains "overlap" by introducing verification domains for analyses.

DEFINITION 1. *An* analysis verification domain σ *is a many-sorted signature:* $(\mathcal{A}, \mathcal{S}, \mathcal{R}, \mathcal{T}, [\![\cdot]\!]_\sigma)$ *where:*

- $\mathcal{A}$ is a set of sorts: $\mathcal{A} = \{A_i\}$.

Examples of sorts are system elements (threads T, CPUs C, and thread scheduling policies $\mathsf{SchedPol}$) and standard sorts (Booleans $\mathcal{B}$ and integers $\mathcal{Z}$).

- $\mathcal{S}$ is a set of static properties: $\mathcal{S} = \{S_i\}$. Each static property is a typed function $S : A_i \times \ldots \times A_j \mapsto A_k$.

Static properties capture analysis-specific system properties set at design-time, such as thread periods ($\mathsf{Per} : T \mapsto \mathcal{Z}$) and processor frequencies ($\mathsf{CPUFreq} : C \mapsto \mathcal{Z}$), and standard operators like addition ($+ : \mathcal{Z} \times \mathcal{Z} \mapsto \mathcal{Z}$).

- $\mathcal{R}$ is a finite set of runtime properties: $\mathcal{R} = \{R_i\}$, $R : A_i \times \ldots \times A_j \mapsto A_k$.

Runtime properties capture system aspects that depend on $\mathcal{S}$ but change during execution, such as preemption between

threads or connectivity between battery cells. Given a runtime state q, $q(R_i)$ means the evaluation of R_i in q. The structure of q is given by $\mathcal{T}$:

- $\mathcal{T}$ is the execution semantics of the verification domain.

Informally, $\mathcal{T}$ is a set of infinite sequences of assignments to the runtime properties. Each sequence corresponds to an execution of the system, and specifically to the values of the runtime properties observed at the successive states of the execution. The executions are infinite since we are interested in analyzing reactive systems which, in general, do not terminate. $\mathcal{T}$ is discussed in detail for specific verification domains in Sec. 4.2.3 and 4.2.4.

- $[\![\cdot]\!]_\sigma$ is a partial interpretation of $\mathcal{A}$ and $\mathcal{S}$. It assigns permissible values to some of the sorts in $\mathcal{A}$ and value mappings to some elements of $\mathcal{S}$.

Intuitively, $[\![\cdot]\!]_\sigma$ interprets common sorts and operators in the standard way. For example $[\![\mathcal{B}]\!] = \mathbb{B}$, $[\![\mathcal{Z}]\!] = \mathbb{Z}$, $[\![+]\!]$ is integer addition, and $[\![\wedge]\!]$ is Boolean conjunction, and so on. In addition, $[\![\cdot]\!]_\sigma$ interprets domain-specific sorts, e.g., the set of permissible scheduling policies $\mathsf{SchedPol}$.

Note that a verification domain σ externalizes concepts of multiple engineering domains to create an *inter-domain* reasoning ground. However, σ does not limit deep *intra-domain* reasoning of each analysis. For example, a thermal runaway analysis would reason about evolving battery temperature, which we do not include into $\mathcal{S}$ or $\mathcal{R}$ since other analyses in our example have no interactions with battery temperature.

Some elements of $\mathcal{A}$ and $\mathcal{S}$ (e.g., T and Per) represent a concrete system design and are left uninterpreted by $[\![\cdot]\!]_\sigma$. To obtain a full interpretation for $\mathcal{A}$, $\mathcal{S}$, and $\mathcal{T}$ we complete $[\![\cdot]\!]_\sigma$ with an *architectural model* (or model, for short).

Definition 2. *An* architectural model M *is an interpretation* $[\![\cdot]\!]_\mathsf{M}$ *of some elements of* $\mathcal{A}$ *and* $\mathcal{S}$*, and* $\mathcal{T}$[1].

For example, a set of three threads and their periods is specified by a model M as $[\![T]\!]_\mathsf{M} = \{t_1, t_2, t_3\}$ and $[\![\mathsf{Per}]\!]_\mathsf{M} = \{t_1 \mapsto 40, t_2 \mapsto 50, t_3 \mapsto 60\}$. The value of runtime properties depends on the execution state, as follows. Let $[\![\cdot]\!]$ denote the combination of $[\![\cdot]\!]_\sigma$ and $[\![\cdot]\!]_\mathsf{M}$. Formally, state q maps each runtime property $R : A_i \times \cdots \times A_j \mapsto A_k$ to a function $q(R) : [\![A_i]\!] \times \cdots \times [\![A_j]\!] \mapsto [\![A_k]\!]$. Let Q be the set of all states, and Q^ω be the set of all infinite sequences of states (i.e., executions). Then $[\![\mathcal{T}]\!]_\mathsf{M} \subseteq Q^\omega$. In other words, $[\![\mathcal{T}]\!]_\mathsf{M}$ is the set of executions of the system defined by M. Note that $\mathcal{R}$ is interpreted indirectly and modally via $[\![\mathcal{T}]\!]_\mathsf{M}$. Specifically, each state q in each execution in $[\![\mathcal{T}]\!]_\mathsf{M}$ gives an interpretation $q(R_i)$ to each $R_i \in \mathcal{R}$.

We require that $[\![\cdot]\!]$ interpret all elements of $\mathcal{A}$, $\mathcal{S}$, and $\mathcal{T}$ unambiguously. In general, analyses from multiple domains are applied to a single model M. Therefore, M must complete the interpretations for $\mathcal{A}$, $\mathcal{S}$, and $\mathcal{T}$ from each domain.

3.2 Contract Language

Our contract language consists of a combination of first-order and temporal logic formulas over the sorts and properties of a target verification domain. We present it in stages, beginning with the syntax of contract formulas.

[1]A classic definition of an architectural model [9] is an annotated graph of components and connectors. Our functional definition is more convenient to specify and verify contracts. Sec. 5.1 explains our infrastructure to transform the two versions of architecture.

3.2.1 Contract Formula Syntax

We first define the static fragment of contract formulas. Consider a domain $\sigma = (\mathcal{A}, \mathcal{S}, \mathcal{R}, \mathcal{T}, [\![\cdot]\!]_\sigma)$. Let V be a denumerable set of typed variables, $V = \{v_i\}, v_i : A_i$. Then the set of static formulas over σ, denoted ϕ, is defined by the following grammar:

$$\phi ::= v \mid f(e_1 \cdots e_j),$$

where $v \in V$, $f \in \mathcal{S}$ with arity j, and $e_1, \ldots, e_j \in \phi$.

Typing. Formulas are well-typed. Variables are typed with their sorts. If the type of f is $A_1 \times \ldots \times A_j \mapsto A_k$, then the type of $f(e_1, \ldots, e_j)$ is A_k and it is well-typed iff the type of each e_i is A_i. For simplicity, we write $v_1 + v_2$ instead of $+(v_1, v_2)$, $e_1 \wedge e_2$ instead of $\wedge\,(e_1, e_2)$, etc.

Temporal Logic Formulas. We now turn our attention to the temporal fragment of contract formulas. Since we are interested in expressing properties of infinite executions of reactive systems, we use a variant of next-time-free linear temporal logic (LTL) [21]. The key difference with standard LTL is that propositions in our logic are not *atomic*, but are constructed from $\mathcal{S}$ and $\mathcal{R}$. The set of all such runtime formulas is defined by the following grammar:

$$RF ::= v \mid f(e_1, \ldots, e_j),$$

where $v \in V$, $f \in \mathcal{S} \cup \mathcal{R}$ with arity j, and $e_1, \ldots, e_j \in \phi$. Runtime formulas are also well-typed in the same way as static formulas. Then our LTL formulas, denoted ψ, are defined by the following grammar:

$$\psi ::= p \mid \neg\psi \mid \psi \wedge \psi \mid \psi\ \mathsf{U}\ \psi,$$

where $p \in RF$ is a runtime formula with Boolean type. Note that other temporal operators, such as G and F (except X, which is not part of the language), are defined in terms of $\neg$, $\wedge$, and U in the standard manner. Finally, we define contract formulas using ϕ and ψ.

Definition 3. *Given a domain* σ*, a set of* contract formulas $\mathcal{F}_\sigma$ *is defined by the following grammar:*

$$\begin{aligned}\mathcal{F}_\sigma ::= {} & \forall\, v_1 \cdots v_j \centerdot \phi \mid \exists\, v_1 \cdots v_j \centerdot \phi \\ & \mid \forall\, v_1 \cdots v_j \centerdot \phi : \psi \mid \exists\, v_1 \cdots v_j \centerdot \phi : \psi,\end{aligned}$$

where ϕ is a static formula of Boolean type, and ψ is a LTL formula – both over variables $v_1, \ldots, v_j$. Thus, a contract formula can be purely first-order (i.e., the first two forms) or a combination of first-order and LTL (i.e., the last two forms). The meaning of the third form is that every assignment of V that satisfies ϕ must also satisfy ψ. The meaning of the last form is that at least one assignment of V satisfies both ϕ and ψ. Note that formulas without variables are also allowed. We are now ready to define analyses and contracts.

3.2.2 Analysis Contracts

Functionally, an analysis An takes an architectural model M_I as input and produces a new architectural model $\mathsf{M}_O = An(\mathsf{M}_I)$ as output. The contract for An specifies restrictions on valid input models and valid output models, as well as the model properties it reads and modifies.

Definition 4. *A* contract C *for an analysis* An *over a verification domain* σ *is a 4-tuple* $C = (I, O, A, G)$*, where:*

- $I \subseteq \mathcal{A} \cup \mathcal{S}$ are sorts and properties read by An.
- $O \subseteq \mathcal{A} \cup \mathcal{S}$ are sorts and properties modified by An.

- $A \subseteq \mathcal{F}_\sigma$ are assumptions – contract formulas that must be satisfied by every valid input model to An.
- $G \subseteq \mathcal{F}_\sigma$ are guarantees – contract formulas over that must be satisfied by every valid output model from An.

In order to determine whether An satisfies its contract, we must first define what it means for An to satisfy a contract formula. This is the topic of the next subsection.

3.2.3 Contract Semantics

We start with the evaluation of static formulas and build up to defining satisfaction of a contract. Let the interpretation of sort A be $[\![A]\!]$ and the interpretation of $f : A_1 \times \cdots \times A_j \mapsto A_k \in \mathcal{S}$ be $[\![f]\!] : [\![A_1]\!] \times \cdots \times [\![A_j]\!] \mapsto [\![A_k]\!]$. An *assignment* μ maps each variable $v : A$ to an element of $[\![A]\!]$. Given a static formula ϕ, and an assignment μ, $[\![\phi, \mu]\!]$ is the evaluation of ϕ under μ defined as:

$$[\![v, \mu]\!] = \mu(v); \quad [\![f(e_1, \ldots, e_j), \mu]\!] = [\![f]\!]([\![e_1, \mu]\!], \ldots, [\![e_j, \mu]\!]).$$

Note that if the type of ϕ is A, then $[\![\phi, \mu]\!] \in [\![A]\!]$. The evaluation of a runtime formula RF under an assignment μ in a state q, denoted $[\![RF, \mu, q]\!]$, is defined as:

$$\begin{aligned} [\![v, \mu, q]\!] &= \mu(v) \\ [\![f(e_1, \ldots, e_j), \mu, q]\!] &= [\![f]\!]([\![e_1, \mu, q]\!], \ldots, [\![e_j, \mu]\!], q),\ f \in \mathcal{S} \\ [\![f(e_1, \ldots, e_j), \mu, q]\!] &= q([\![f]\!])([\![e_1, \mu, q]\!], \ldots, [\![e_j, \mu, q]\!]),\ f \in \mathcal{R} \end{aligned}$$

An execution $\pi = q_0, q_1, \ldots$ satisfies an LTL formula ψ under assignment μ, denoted $\pi, \mu \models \psi$, if:

- If $\psi \in RF$ then $\pi, \mu \models \psi$ iff $[\![\psi, \mu, q_0]\!] = \top$.
- $\pi, \mu \models \neg\psi'$ iff $\pi, \mu \not\models \psi'$.
- $\pi, \mu \models \psi_1 \wedge \psi_2$ iff $\pi, \mu \models \psi_1$ and $\pi, \mu \models \psi_2$.
- $\pi, \mu \models \psi_1 \ \mathsf{U}\ \psi_2$ iff there exists $i \geq 0$ such that for all $0 \leq j < i$. $\pi^j, \mu \models \psi_1$ and $\pi^i, \mu \models \psi_2$, where π^x is the (infinite) suffix of π starting with state q_x.

Consider a model M. Let $[\![\cdot]\!]$ be the combination of $[\![\cdot]\!]_\sigma$ and $[\![\cdot]\!]_\mathsf{M}$. A contract formula $f \in \mathcal{F}_\sigma$ is satisfied by M, denoted $\mathsf{M} \models f$, based on the form of f (see Def. 3) as follows:

- f is of the first form and $\forall\, \mu \in \mathcal{V}$. $[\![f, \mu]\!] = \top$.
- f is of the second form and $\exists\, \mu \in \mathcal{V}$. $[\![f, \mu]\!] = \top$.
- f is of the third form and $\forall\, \mu \in \mathcal{V} . \forall\, \pi \in [\![\mathcal{T}]\!]$. $[\![\phi, \mu]\!] = \top \Rightarrow \pi, \mu \models \psi$.
- f is of the fourth form and $\exists\, \mu \in \mathcal{V} . \forall\, \pi \in [\![\mathcal{T}]\!]$. $[\![\phi, \mu]\!] = \top \wedge \pi, \mu \models \psi$.

Thus, first-order quantification over static formulas is interpreted in the usual way: universal quantification over a static formula and an LTL formula holds if and only if all assignments that satisfy the static formula also satisfy the LTL formula; existential quantification over a static formula and an LTL formula holds if and only if there exists an assignment that satisfies both static and LTL formulas.

Analysis Applicability. A model to which an analysis is applied should satisfy its assumption, and the resulting model should satisfy its guarantee. Formally:

Name	Type	Description
Per	$T \mapsto \mathcal{Z}$	Thread's period.
Dline	$T \mapsto \mathcal{Z}$	Thread's deadline.
WCET	$T \mapsto \mathcal{Z}$	Thread's worst case execution time.
ThSecCl	$T \mapsto$ SecCl	Thread's security class.
CPUSchedPol	$C \mapsto$ SchedPol	CPU's scheduling policy.
CPUFreq	$C \mapsto \mathcal{R}$	CPU's normalized frequency[2].
NotColoc	$T \mapsto 2^T$	Thread t mapped to a set of threads that should not share the same CPU as t.
CPUBind	$T \mapsto C$	Thread-to-CPU binding.
ThSafe	$C \mapsto \mathcal{B}$	Flag whether CPU's threads are thread-safe.
Voltage	$() \mapsto \mathcal{R}$	Required system voltage[3].

Table 1: Static properties of σ_{Sched}.

DEFINITION 5. *Analysis An with contract $An.C = (I, O, A, G)$ is* applicable *to model* M*, denoted* $\mathsf{M} \models An.C$*, iff* $\forall\, a \in A$. $\mathsf{M} \models a$ *and* $\forall\, g \in G$. $An(\mathsf{M}) \models g$.

In Sec. 4 we present an algorithm to decide $\mathsf{M} \models C$. Now, to highlight our approach, we turn to formalizing analysis domains and contracts for the example from Sec. 2.

3.3 Scheduling Verification Domain

The scheduling verification domain σ_{Sched} encodes the semantics of the real-time scheduling of threads along with their allocation to processors. This domain helps specify contracts for analyses that decide valid thread allocations and priority assignments, check schedulability according to a selected scheduling policy, determine processor frequency, etc. The domain is defined as $\sigma_{Sched} = (\mathcal{A}_{Sched}, \mathcal{S}_{Sched}, \mathcal{R}_{Sched}, \mathcal{T}_{Sched}, [\![\cdot]\!]_{Sched})$. In addition to Booleans $\mathcal{B}$ and integers $\mathcal{Z}$, $\mathcal{A}_{Sched}$ has reals $\mathcal{R}$, threads T, CPUs C, thread security classes SecCl, and thread scheduling policies SchedPol. Sorts $\mathcal{B}$, $\mathcal{Z}$, $\mathcal{R}$ are interpreted by $[\![\cdot]\!]_{Sched}$ in a standard way. There are three security classes – normal, secret, and top-secret – $[\![\mathsf{SecCl}]\!]_{Sched} = \{\mathbf{normal}, \mathbf{secret}, \mathbf{topsecret}\}$ and three scheduling policies – rate monotonic scheduling (RMS), earliest deadline first (EDF) [18], and deadline monotonic scheduling (DMS) [1] – $[\![\mathsf{SchedPol}]\!]_{Sched} = \{\mathbf{rms}, \mathbf{dms}, \mathbf{edf}\}$. Sorts T and C are uninterpreted since they are model-dependent.

In addition to Boolean, integer, and real arithmetic operators, the properties in $\mathcal{S}_{Sched}$ interpreted by M are summarized in Tab. 1. Also, there is one runtime property $\mathcal{R}_{Sched} = \{\mathsf{CanPrmpt} : T \times T \mapsto \mathcal{B}\}$ such that $q(\mathsf{CanPrmpt}(t_1, t_2)) = \top$ iff t_1 can preempt t_2 in state q. Interpretation of $\mathcal{T}_{Sched}$ and CanPrmpt is model-dependent and presented in Sec. 4.

3.4 Battery Verification Domain

The domain of battery design and usage $\sigma_{Batt} = (\mathcal{A}_{Batt}, \mathcal{S}_{Batt}, \mathcal{R}_{Batt}, \mathcal{T}_{Batt}, [\![\cdot]\!]_{Batt})$ is defined as follows. Sorts $\mathcal{B}, \mathcal{Z}, \mathcal{R} \in \mathcal{A}_{Batt}$ and their interpretations are identical to σ_{Sched}. The set of batteries is $B \in \mathcal{A}_{Batt}$, and $\mathsf{ConnSchedPol} \in \mathcal{A}_{Batt}$ is the set of three battery scheduling policies [12][17]: unweighed round robin with fixed cell groups (**FGuRR**), weighed kRR with fixed parallel cell groups (**FGwRR**), and weighed kRR with cell group packing (**GPwRR**). Voltage $\in \mathcal{S}_{Batt}$ is the same as in the scheduling domain: our approach naturally captures the fact

[2] A real number between 0 and 1.

[3] Voltage is a nullary function, or a real constant. We consider a simplified example where the system voltage is the maximum of required individual processor voltages.

Name	Type	Description
Voltage	$() \mapsto \mathcal{R}$	Required system voltage.
BatRows	$B \mapsto \mathcal{Z}$	Battery's cell rows.
BatCols	$B \mapsto \mathcal{Z}$	Battery's cell columns.
BatConnSchedPol	$B \mapsto$ ConnSchedPol	Battery's cell scheduling policy.
SerialReq	$B \mapsto \mathcal{Z}$	Number of cells required to connect in series[5].
ParalReq	$B \mapsto \mathcal{Z}$	Number of cells required to connect in parallel[6].
K	$B \times \mathcal{Z} \mapsto \mathcal{Z}$	Weight of cells with i thermal neighbors.
HasReqdLifetime	$B \mapsto \mathcal{B}$	Flag whether a battery has the lifetime required.

Table 2: Static properties of σ_{Batt}.

that both domains are concerned with the system voltage. $\mathcal{S}_{Batt}$ contains the properties summarized in Tab. 2.

Informally, a battery execution consists of continuous charging, discharging, and resting of cells. The precise semantics $[\![\mathcal{T}_{Batt}]\!]$ is model-dependent and defined in Sec. 4. There is one runtime property: $\mathcal{R}_{Batt} = \{\mathsf{TN} : B \times \mathcal{Z} \to \mathcal{Z}\}$. When a battery b is in state q, $q(\mathsf{TN}(b, i))$ denotes the number of cells with i *thermal neighbors* – cells that exchange heat conductively through a connector [4]. This is motivated by results [16]: there is a close connection between thermal neighbors and thermal runaway. Specifically, there exist constants $\mathsf{K}(b, i) : b \in B, i \in \mathbb{Z}$ such that a state q triggers a thermal runaway in battery b if it violates the condition:

$$\sum_i \mathsf{K}(b, i) \times q(\mathsf{TN}(b, i)) \geq 0 \tag{1}$$

However, the exact values of K are not known up-front, and experiments with a battery are needed to obtain them.

3.5 Analysis Contracts for Running Example

Using the domains signatures σ_{Sched} and σ_{Batt} we define the contracts for analyses $\mathcal{AN}$ from Sec. 2.

Secure thread allocation[7] ($An_{SecAlloc}$) has contract $C_{SecAlloc} : I = \{T, \mathsf{ThSecCl}\}$, $O = \{\mathsf{NotColoc}\}$, $A = \varnothing$, and $G = \{g\}$ where g is:

$$\forall\, t_1, t_2 \,.\, \mathsf{ThSecCl}(t_1) \neq \mathsf{ThSecCl}(t_2) \Rightarrow t_1 \in \mathsf{NotColoc}(t_2)$$

Thus, $An_{SecAlloc}$ makes no assumptions, but guarantees that threads with different security classes are never co-located. We omit sorts, e.g., $t_1 : T$ and $t_2 : T$, since they are implied by typing rules.

Bin packing ($An_{BinPack}$) has contract $C_{BinPack}$: $I = \{T, C, \mathsf{NotColoc}, \mathsf{Per}, \mathsf{WCET}, \mathsf{Dline}\}$, $O = \{\mathsf{CPUBind}\}$, $A = \varnothing$, and $G = \{g\}$ where g is:

$$\forall\, t_1, t_2 \,.\, t_1 \in \mathsf{NotColoc}(t_2) \Rightarrow \mathsf{CPUBind}(t_1) \neq \mathsf{CPUBind}(t_2)$$

Thus, $An_{BinPack}$ makes no assumptions but guarantees that threads that should not be co-located are never scheduled on the same CPU.

Frequency scaling (An_{FreqSc}) has contract C_{FreqSc}: $I = \{T, C, \mathsf{CPUBind}, \mathsf{Dline}\}$, $O = \{\mathsf{CPUFreq}\}$, $G = \varnothing$, and

$$A \triangleq \{\forall\, t_1, t_2 \,.\, t_1 \neq t_2 \wedge \mathsf{CPUBind}(t_1) = \mathsf{CPUBind}(t_2) : \mathsf{G}\ (\mathsf{CanPrmpt}(t_1, t_2) \Rightarrow \mathsf{Dline}(t_1) < \mathsf{Dline}(t_2))\}$$

Thus, An_{FreqSc} makes no guarantees but assumes that the scheduling used is semantically equivalent to a deadline-monotonic scheduling policy. Note that a scheduling policy can be DMS for a specific model, e.g., rate-monotonic scheduling (RMS) for a harmonic model, even thought it is not deadline monotonic in general.

Model checking with REK (An_{REK}) has contract C_{REK}: $I = \{T, C, \mathsf{Per}, \mathsf{Dline}, \mathsf{WCET}, \mathsf{CPUBind}\}$, $O = \{\mathsf{ThSafe}\}$, $G = \varnothing$, and $A = \{a_1, a_2\}$ where:

$$a_1 \triangleq \forall\, t \,.\, \mathsf{Per}(t) = \mathsf{Dline}(t),$$

$$a_2 \triangleq \forall\, t_1, t_2 \,.\, \mathsf{G}\ (\mathsf{CanPrmpt}(t_1, t_2) \Rightarrow \mathsf{G}\ \neg\mathsf{CanPrmpt}(t_2, t_1)).$$

REK [4] takes threads and their marked source code files (which we didn't include into the formal example) as input and verifies whether the system is safe, where safety is expressed as assertions embedded in the source code. An_{REK} assumes implicit deadlines and fixed-priority scheduling. Prior to this work, the only way to apply REK was to use RMS. However, our contract mechanism allows for a broader scope of applicability. Note that a_1 expresses implicit deadlines, while a_2 expresses fixed priority scheduling, i.e., if t_1 preempts t_2, then t_2 should never be able to preempt t_1.

Thermal runaway ($An_{ThermRun}$) has contract $C_{ThermRun}$: $I = \{B, \mathsf{BatRows}, \mathsf{BatCols}, \mathsf{Voltage}\}$, $O = \{\mathsf{K}\}$, $A = \varnothing$, and $G = \varnothing$. Note that $An_{ThermRun}$ has no assumptions or guarantees, but has a dependency with defined below battery scheduling via I and O. Thermal runaway determines the patterns, which, given concrete battery characteristics, would result into a thermal runaway. In our example, we encode these patterns as $\mathsf{K}(i)$ for $i : \mathcal{Z} \in [0, 3]$. $An_{ThermRun}$ determines K through experimentation, adjusting K so that acceptable heat propagation patterns satisfy (1), and unacceptable ones violate it.

Battery scheduling ($An_{BatSched}$) has contract $C_{BatSched} : I = \{B, \mathsf{BatRows}, \mathsf{BatCols}\}$, $O = \{\mathsf{BatConnSchedPol}, \mathsf{HasReqdLifetime}, \mathsf{SerialReq}, \mathsf{ParalReq}\}$, $A = \varnothing$, and $G = \{g\}$ where g is:

$$\forall\, b \,.\, \mathsf{G}\ (\mathsf{K}(b, 0) \times \mathsf{TN}(b, 0) + \mathsf{K}(b, 1) \times \mathsf{TN}(b, 1) + \mathsf{K}(b, 2) \times \mathsf{TN}(b, 2) + \mathsf{K}(b, 3) \times \mathsf{TN}(b, 3) \geq 0).$$

$An_{BatSched}$ computes a battery cell connectivity scheduler that maximizes the battery lifetime given the battery characteristics and output requirements. It sets a flag indicating whether the battery with the selected scheduler meets the lifetime requirement. Since the scheduling is not aware of the thermal runaway, the determined scheduler needs to be verified against the thermal runaway pattern, hence the guarantee. $An_{BatSched}$ also sets cell group characteristics SerialReq and ParalReq that are used to verify its guarantee.

4. CONTRACT VERIFICATION

This section presents our contract verification algorithm, which takes a model M and a set of analyses $\mathcal{AN}$ and produces a correct execution of $\mathcal{AN}$ on M, or aborts. The algorithm consists of the following steps: (i) determine an ordering $\mathcal{O}$ of $\mathcal{AN}$ that respects all inter-analysis dependencies; (ii) process each analysis $An \in \mathcal{AN}$ with contract $C = (I, O, A, G)$ in the order $\mathcal{O}$ by: (iia) verifying

[4] As opposed to electrical neighbors – cells that are connected to each other electrically, no matter how far apart physically they are.

[5] SerialReq is a battery-specific form of the voltage output requirement.

[6] ParalReq is a battery-specific form of the electrical current output requirement.

[7] We omit implied atoms in inputs, like SecCl in $C_{SecAlloc}.I$.

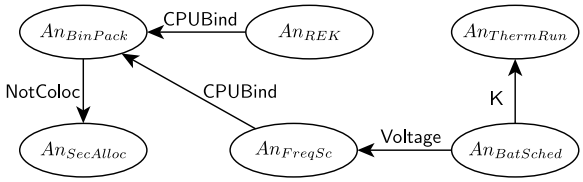

Figure 1: Analyses graph for the running example.

$\forall\, a \in A \,.\, \mathsf{M} \models a$, (iib) updating M by executing An, i.e., setting M to $An(\mathsf{M})$, (iic) and verifying $\forall\, g \in G \,.\, \mathsf{M} \models g$; and (iii) output the final M as the result. The algorithm aborts if either an appropriate ordering $\mathcal{O}$ cannot be computed (as we will show, it only happens if there are circular dependencies between analyses), if any of the verifications in Steps (iia) and (iic) fail, or the execution of An on M fails. Note that the algorithm ensures that all analyses produce valid results since: (a) an analysis An executes successfully on M only if $\mathsf{M} \models An.C$ (see Def. 5); and (b) the ordering $\mathcal{O}$ ensures that once an analysis has been executed, future values of M do not violate its assumptions.

4.1 Analysis Ordering

We begin with Step (i), which uses contracts to determine a correct ordering of analyses execution. The contract of analysis An is denoted $C(An)$. For a contract $C = (I, O, A, G)$, $C.I$ means I, $C.O$ means O, etc.

Definition 6. *Analysis contract C_i* is dependent *on analysis contract C_j, denoted $d(C_i, C_j)$, if $C_i.I \cap C_j.O \neq \varnothing$. Analysis An_i* is dependent *on analysis An_j, denoted $d(An_i, An_j)$, iff $d(C(An_i), C(An_j))$.*

Given a set of analyses $\mathcal{AN}$, an ordering $\mathcal{O} = \langle An_1 \cdots An_n \rangle$ of $\mathcal{AN}$ is *sound* if each analysis in the ordering is not dependent on any of its predecessors, i.e.,

$$\forall\, i \in [1, n] \,.\, \forall\, j \in [1, i) \,.\, \neg d(An_i, An_j)$$

Thus, the goal of step (i) is to produce a sound ordering of $\mathcal{AN}$. Consider the directed graph of analyses $\gamma = (\mathcal{AN}, d(,))$. It can be shown that: (i) if γ is cyclic then there is no sound ordering of $\mathcal{AN}$; and (ii) otherwise γ is a DAG and any topologically sorted ordering of its nodes is a sound ordering of $\mathcal{AN}$. Therefore, $\mathcal{O}$ is computed by: (i) constructing γ; (ii) checking its cyclicity; (iii) aborting if γ is cyclic; and (iv) if it is acyclic, constructing any topological ordering of its nodes. The specific $\mathcal{O}$ constructed does not affect the correctness of our algorithm since topological orderings of γ differ only in relative positions of mutually independent analyses. The γ for our example is shown in Fig. 1. Each edge is labeled by a property that causes the dependency between the corresponding analyses.

4.2 Analysis Applicability

We now focus on Steps (iia) and (iic) of our algorithm. Recall that the core problem here is to decide $\mathsf{M} \models f$ where f is a contract formula. In Step (iia) f is one of the contract's assumptions, while the Step (iic) it is one of the contract's guarantees. The algorithm for verifying $\mathsf{M} \models f$ depends on the form of f (see Def. 3), and we consider each separately.

4.2.1 Verifying Purely First Order Formulas

If f is a quantified first-order formula (i.e., the first two forms of Def. 3) we check $\mathsf{M} \models f$ via Satisfiability Modulo Theories (SMT) solving. We first describe how to construct a SMT formula $\varphi(\mathsf{M}, \phi)$ – using SMT v2 syntax [7] – given a model M and a first-order formula ϕ. The first step in constructing $\varphi(\mathsf{M}, \phi)$ is defining the basic types. These are obtained directly from the sorts ($\mathcal{A}$) of the analysis domain. Basic sorts like Boolean, integer, and float are already primitive types in SMT. Domain-specific sorts, like threads and processors, are declared as integers using the `define-sort` SMT command. Subsequently, the IDs of the actual threads and processors are used as concrete values for their corresponding types. For example, if M has five threads with IDs $[0, 4]$ and three processors with IDs $[0, 2]$, then $\varphi(\mathsf{M}, \phi)$ has two types – `thread` and `processor` – defined as follows:

```
(define-sort thread() (Int))
(define-sort processor() (Int))
```

Subsequently, all variables of type T have five legal values $[0, 4]$ and all variables of type C have three legal values $[0, 2]$. These legal values are enforced by adding appropriate SMT constraints to $\varphi(\mathsf{M}, \phi)$. The functions in $\varphi(\mathsf{M}, \phi)$ are obtained from the static properties of the domain used for the contract. For example, if the analysis references Per, then $\varphi(\mathsf{M}, \phi)$ contains a function `Period` defined as follows:

```
(declare-fun Period (thread) Int)
```

Variables are also declared with appropriate types, e.g., variable $t_1 : T$ is declared as: `(declare-fun t1 () (thread))`. Finally, the constraints in $\varphi(\mathsf{M}, \phi)$ are obtained from: the interpretation of static properties of the domain (e.g., if thread 0 has a period 20, then we add the constraint `(assert (= (period 0) 20))`) and the formula ϕ itself (e.g., if $\phi \triangleq \mathsf{Per}(t1) < \mathsf{Per}(t2)$, then we add the constraint `(assert (< (period t1) (period t2)))`.

Given an SMT formula φ as input, an SMT solver returns SAT if φ is satisfiable, and UNSAT otherwise. Now checking $\mathsf{M} \models f$ reduces to two cases:

- $f = \forall\, v_1 \cdots v_j \,.\, \phi$ (form 1 of Def. 3): In this case we return YES if the SMT solver returns UNSAT for input $\varphi(\mathsf{M}, \neg\phi)$ and NO otherwise.
- $f = \exists\, v_1 \cdots v_j \,.\, \phi$ (form 2 of Def. 3): In this case we return YES if the SMT solver returns SAT for input $\varphi(\mathsf{M}, \phi)$ and NO otherwise.

The correctness of our algorithm follows from our semantics and the construction of $\varphi(\mathsf{M}, \phi)$. An example of $\varphi(\mathsf{M}, \phi)$ constructed for $C_{BinPack}.G$ is shown in Fig. 2. The guarantee states that non-colocated threads should be bound to different processors, and has the first form of Def. 3. The SMT solver returns UNSAT, hence $\mathsf{M} \models C_{BinPack}.G$.

4.2.2 Verifying First Order+LTL Formulas

We now show how to verify f if it combines both first-order logic and LTL (i.e., the last two forms of Def. 3). This again has two cases:

- $f = \forall\, v_1 \cdots v_j \,.\, \phi : \psi$ (form 3 of Def. 3):

We first construct $\varphi(\mathsf{M}, \phi)$. Next, we use the SMT solver iteratively to compute all satisfying solutions of $\varphi(\mathsf{M}, \phi)$. To obtain all solutions, we use "blocking clauses", i.e., once we obtain a solution, we add its negation to the formula before re-solving it. From each solution we construct the corresponding assignment μ to $\{v_1 \cdots v_j\}$. For each assignment

```
(define-sort thread () (Int))
(define-sort processor () (Int))
(declare-fun Actual_Processor_Binding (thread) Int)
(define-fun Not_Colocated((x1 thread) (x2 thread)) Bool
  (ite (and (= x1 0)(= x2 1)) true (ite (and (= x1 0)(= x2 2)) true
  (ite (and (= x1 1)(= x2 0)) true (ite (and (= x1 2)(= x2 0)) true   false)))))
(assert (= (Actual_Processor_Binding 0) 0))
(assert (= (Actual_Processor_Binding 1) 1))
(assert (= (Actual_Processor_Binding 2) 1))
(assert (not (forall ((x1 thread) (x2 thread)) (=>
    (and (or (= x1 0) (= x1 1) (= x1 2))
      (or (= x2 0) (= x2 1) (= x2 2)))
    (=> (and (not (= x1 x2)) (Not_Colocated x1 x2))
      (not (= (Actual_Processor_Binding x1) (Actual_Processor_Binding x2))))))))
(check-sat)
```

Figure 2: SMT problem for verifying $C_{BinPack}.G$.

μ, we check $\forall \pi \in [\![\mathcal{T}]\!] \centerdot \pi, \mu \models \psi$ using a model checker. The model checking step is domain-specific and described in the following subsections. We return YES if the model checker finds no ψ violations for every μ, and NO otherwise.

- $f = \exists\, v_1 \cdots v_j \centerdot \phi : \psi$ (form 4 of Def. 3):

We first construct $\varphi(\mathsf{M}, \phi)$. Next, we use the SMT solver iteratively to compute all assignments μ to $\{v_1 \cdots v_j\}$ as in the previous case. For each assignment μ, we check $\forall \pi \in [\![\mathcal{T}]\!] \centerdot \pi, \mu \models \psi$ using a model checker. We return YES if the model checker find no ψ violations for at least one μ, and NO otherwise. The correctness of our algorithm follows from our semantics and the construction of $\varphi(\mathsf{M}, \phi)$. The algorithm always terminates if the sort of each quantified variable v_i is interpreted to a finite domain (e.g., threads and batteries, but not integers), since this means that there are only a finite set of assignments to $\{v_1 \cdots v_j\}$. This is indeed the case for all analyses in our example. We now describe the model checking step for the scheduling and battery domains.

4.2.3 *Model Checking for Scheduling*

The execution semantics $[\![\mathcal{T}_{Sched}]\!]$ of the scheduling domain is defined as follows. Recall that each thread consists of an infinite and periodic sequence of jobs. A state q of the system corresponds to a point in time where a new job arrives or a currently executing job terminates. An execution consists of a infinite sequence of such states observed at runtime. Note that multiple executions are possible due to the non-determinism in the time required by each job to complete. Then $[\![\mathcal{T}_{Sched}]\!]$ consists of all such executions.

We model $[\![\mathcal{T}_{Sched}]\!]$ as a *Kripke structure* $\mathcal{K}([\![\mathcal{T}_{Sched}]\!])$ composed of a "task" process for each thread. Task processes are periodic and their numeric characteristics – $(\mathsf{Per}, \mathsf{Dline}, \mathsf{WCET})$ – are specified by the model M. There are $|[\![C]\!]_{\mathsf{M}}|$ processors, and each running task is allocated to a processor dynamically. For each task process t, $\mathcal{K}([\![\mathcal{T}_{Sched}]\!])$ has the following propositions:

- $\mathsf{Prior}(t) : \mathcal{Z}$ – the priority of t.
- $\mathsf{Run}(t) : \mathcal{B}$ – whether a job of t is dispatched on a processor.
- $\mathsf{InQ}(t) : \mathcal{B}$ – whether a job of t has arrived but hasn't been completed yet.

$\mathsf{Prior}(t)$ is set by the scheduling policy and decides which tasks are executed. The last two propositions encode every possible state of t: idle if $\neg\mathsf{InQ}(t) \wedge \neg\mathsf{Run}(t)$, waiting for processor if $\mathsf{InQ}(t) \wedge \neg\mathsf{Run}(t)$, and executing if $\mathsf{InQ}(t) \wedge \mathsf{Run}(t)$. Also, for any state q of $\mathcal{K}([\![\mathcal{T}_{Sched}]\!])$, and threads t_1, t_2, $q(\mathsf{CanPrmpt})(t_1, t_2)$ is $\top$ iff the following holds in q:

$$\mathsf{Run}(t_1) \wedge \neg\mathsf{Run}(t_2) \wedge \mathsf{InQ}(t_2)$$

Recall that our model checking problem is $\forall \pi \in [\![\mathcal{T}_{Sched}]\!] \centerdot \pi, \mu \models \psi$, where μ is a variable assignment. We solve this by: (i) instantiating the LTL formula ψ to a propositional LTL formula ψ_{prop} using μ; and (ii) using a model checker to verify $\mathcal{K}([\![\mathcal{T}_{Sched}]\!]) \models \psi_{prop}$. For example, $C_{FreqSc}.A$ is expressed as the following propositional LTL formula:

$$\mathsf{G}\ (\mathsf{Run}(\mu(t_1)) \wedge \neg\mathsf{Run}(\mu(t_2)) \wedge \mathsf{InQ}(\mu(t_2)) \Rightarrow \mathsf{Dline}(\mu(t_0)) \leq \mathsf{Dline}(\mu(t_1)))$$

The correctness of our algorithm follows from the semantics of our LTL formulas, the semantics of propositional LTL, and the correctness of model checking.

4.2.4 *Model Checking for Battery*

Let us now define the execution semantics of $[\![\mathcal{T}_{Batt}]\!]$. A battery b consists of a matrix of cells χ being continuously charged, discharged, connected, and disconnected with each other. A state q of the system corresponds to a point in time when either the charge or the connectivity status of a cell changes. An execution consists of an infinite sequence of such states observed at runtime. Many such executions are possible due to the non-determinism in the order of charge and discharge. Then $[\![\mathcal{T}_{Therm}]\!]$ consists of all such executions.

We model $[\![\mathcal{T}_{Therm}]\!]$ as a Kripke structure $\mathcal{K}([\![\mathcal{T}_{Therm}]\!])$ with the following propositions for each cell $c = (x, y) \in \chi$, which is characterized by its physical coordinates $x \in [0, \mathsf{BatRows}-1]$ and $y \in [0, \mathsf{BatCols}-1]$:

- $\mathsf{CellCharge}(c)$ is the charge of c. To simplify model checking we chose a Boolean abstraction for the cell charge, but other abstractions are possible too.
- $\mathsf{CellSt}(c)$ is the status of c with possible values **discharging**, **charging**, and **idle**.
- $\mathsf{Gr}(c)$ is the number of group of cells electrically connected in serial within which c is located. Groups are treated as electrically connected in parallel with each other. Every cell belongs to a group, but not every group or cell is discharging.

TN is encoded as follows. Cells c_1 and c_2 are thermal neighbors, denoted $istnbr(c_1, c_2)$, if: (i) $c_1 \neq c_2$; (ii) $\mathsf{Gr}(c_1) = \mathsf{Gr}(c_2)$; (iii) $|c_1.x - c_2.x| + |c_1.y - c_2.y| \leq TNDIST$ [8]; (iv) $\mathsf{CellCharge}(c_1) = \mathsf{CellCharge}(c_2) = \top$; and (v) $\mathsf{CellSt}(c_1) = \mathsf{CellSt}(c_2) =$ **discharging**. The number of thermal neighbors of cell c is $ntnbr(c) = |\{c' \in \chi \centerdot istnbr(c, c')\}|$. Finally, $\mathsf{TN}(b, i) = |\{c' \in \chi \centerdot ntnbr(c) = i\}|$.

5. IMPLEMENTATION

In this section we present the implementation of our analysis contracts tool[9] in OSATE – an open source environment for AADL modeling [9]. Below we describe how our tool implements the concepts of this paper and how Spin/Promela is used for model checking.

[8]For our calculations we use $TNDIST = 2$.

[9]Available at `www.cs.cmu.edu/~iruchkin/dist/v0.1-emsoft14-tool.tar.gz`.

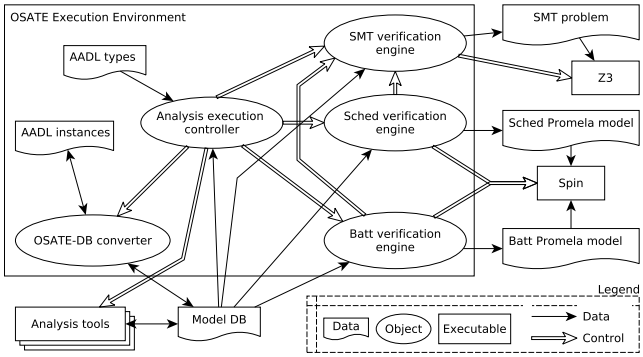

Figure 3: Contracts framework architecture.

5.1 Contracts Framework

Let us start by describing how the different elements of our approach are represented. First, architectural model M is described in an AADL model. Second, domain types, like SecCl and ConnSchedPol, are defined as AADL property types. Third, component sorts T, C, and B are derived from respectively threads, CPUs, and battery devices in the AADL model. $\mathcal{S}_{Sched}$ and $\mathcal{S}_{Batt}$ are derived from properties of the components. Finally, we specify contracts in an AADL sub-language *annex* to capture I, O, A, and G.

Fig. 3 depicts the architecture of our tool. Analysis contracts $\mathcal{C}$ are associated with AADL component types, while M is derived from the AADL main system instance. Initially, our tool converts M from AADL into a database representation using the *OSATE-database converter*. All subsequent steps are performed using this database (model DB). The *analysis execution controller* constructs the analysis graph γ, as described in Sec. 4.1, and delegates the verification of A and G to an appropriate *verification engine*, which is determined by the form in Def. 3, along with $\mathcal{A}$, $\mathcal{T}$, and $\mathcal{R}$ in the contract. A selected verification engine populates an SMT problem and a Promela model with values from the model DB, executes the verification via Z3 and Spin, and interprets the outputs. To verify forms 3 and 4 of Def. 3, σ_{Sched} and σ_{Batt} verification engines use Z3 to find assignments to $v_1 \cdots v_j$ that satisfy ϕ, and use these assignments to verify ψ via Spin.

5.2 Scheduling Domain Implementation

We encode $\mathcal{K}([\![\mathcal{T}_{Sched}]\!])$ as a Promela (the input language of the Spin [13]) program that computes $q(\mathsf{CanPrmpt})(t_1, t_2)$ appropriately for each state q and pair of threads t_1 t_2, as described in Sec. 4.2.3. We implement each task t as a process and add a manager process that decides what priorities are assigned to threads and what threads are dispatched to processors. Thus, the manager process plays the role of a scheduler and a dispatcher. Our Promela program handles the events of job arrival and termination in an infinite cycle, interleaving each event with the manager execution.

The Promela program models non-deterministic job terminations without explicit time counters as follows. The manager process calculates possible upcoming events. Time is advanced in a greedy manner (i.e., whenever possible): if an arrival event happens, or the earliest of all the possible job termination events. To achieve a finite state space, all clock variables[10] are reduced by the minimum value of all clock variables periodically. It can be shown that this approach maintains contract satisfaction as long as clock variables are not used in a contract. This condition holds for our contract language since the clock variables are not exposed in it.

5.3 Battery Domain Implementation

We encode $\mathcal{K}([\![\mathcal{T}_{Batt}]\!])$ as a single-process Promela program. The program maintains the state $\mathsf{CellCharge}, \mathsf{CellSt}, \mathsf{Gr}$ discussed in Sec. 4.2.4. The program execution works in two steps: scheduling cells for discharge and charge (i.e., changing Gr and CellSt), and advancing the charge state (i.e., changing $\mathsf{CellCharge}$).

The first step is deterministic: it imitates the logic of a selected scheduler. **FGURR** does not change Gr and rotates through groups, setting $\mathsf{ParalReq}$ groups to discharge each time and the rest to idle. **FGWRR** does not change Gr either, but instead of rotating the groups it sorts them in decreasing order of charge (which, for us, is the number of cells with $\mathsf{CellCharge}(c) = \top$) and selects the top $\mathsf{ParalReq}$ groups. **GPWRR** assembles groups by packing as many charged cells into each group as possible. Then it selects the top $\mathsf{ParalReq}$ most charged groups to discharge. Within each group, all schedulers select $\mathsf{SerialReq}$ charged cells.

The other step is non-deterministic: every discharging cell non-deterministically becomes discharged; every charging cell non-deterministically becomes charged; idle cells, however, do not change their charges. The program terminates when there is not enough charge for the output requirements.

This program is an overapproximation of high-fidelity battery models with precise measurements of the cell charge. This measurement is then used in these models to schedule the cells. Thanks to the non-determinism in the second step, our implementation accounts for possible cell failures (i.e., cell gets immediately discharged) and any high-fidelity model of charge. On the other hand, the program represents cell schedulers' logic precisely.

6. EVALUATION

In this section we evaluate three aspects of our approach: (i) the effective integration of analyses – detection of integration errors or demonstration of their absence, (ii) the scalability of our tool for models of practical size, and (iii) extensibility of our tool.

Effectiveness of Analysis Integration. Consider a concrete configuration for the sample aircraft: threads t_1, t_2, t_3 have $[\![\mathsf{Per}]\!]_{\mathsf{M}} = \{t_1 \mapsto 100, t_2 \mapsto 150, t_3 \mapsto 200\}$, $[\![\mathsf{Dline}]\!]_{\mathsf{M}} = \{t_1 \mapsto 100, t_2 \mapsto 90, t_3 \mapsto 200\}$, $[\![\mathsf{WCET}]\!]_{\mathsf{M}} = \{t_1 \mapsto 10, t_2 \mapsto 15, t_3 \mapsto 20\}$ are allocated to a single CPU. Before analysis An_{FreqSc} is applied to determine CPU frequencies, its assumption $C_{FreqSc}.A$ is verified. Recall that $C_{FreqSc}.A$ states that the scheduling policy must be semantically equivalent to DMS. Suppose first that the system uses RMS scheduling, i.e., $\mathsf{Prior}(t_1) > \mathsf{Prior}(t_2) > \mathsf{Prior}(t_3)$. In this case, our tool detects a violation of $C_{FreqSc}.A$ via model checking because in this case DMS would assign $\mathsf{Prior}(t_2) > \mathsf{Prior}(t_1)$. Now suppose that the system uses EDF. As our Spin program indicates, it satisfies $C_{FreqSc}.A$ Thus, our approach not only prevents incorrect usage of An_{FreqSc}, but also extends its applicability to EDF.

Next, suppose our system has a battery with $\mathsf{BatRows} =$

[10]Such as the next job arrival or the absolute system time.

Threads	DMS/RMS Time[12]	EDF Time
3	0.01	0.01
4	0.01	0.52
5	0.07	33.4
6	0.37	2290.0
7	2.18	MEMLIM
8	12.4	MEMLIM
9	71.2	MEMLIM
10	421	MEMLIM
11	MEMLIM	MEMLIM

Table 3: Scalability of the $\mathcal{K}([\![\mathcal{T}_{Sched}]\!])$ program.

Cells	FGURR Time [12]	FGWRR Time	GPWRR Time
9	0.13	0.15	0.15
12	0.61	2.34	3.94
16	44.0	31.4	127
20	1060	619	MEMLIM
25	MEMLIM	MEMLIM	MEMLIM

Table 4: Scalability of the $\mathcal{K}([\![\mathcal{T}_{Batt}]\!])$ program.

$\mathsf{BatCols} = 4$, and a voltage requirement $\mathsf{ParalReq} = \mathsf{SerialReq} = 3$. It has been observed [16] that heat-dissipating cells (i.e., those with many thermal neighbors) and heat-isolated cells (i.e., those with no thermal neighbors) tend to prevent thermal runaway, while cells with one thermal neighbor tend to accumulate heat and lead to runaway. An assignment of weights $\mathsf{K}(0) = \mathsf{K}(1) = \mathsf{K}(2) = 2, \mathsf{K}(1) = -1$ in (1) captures this intuition. After executing analysis $An_{BatSched}$, which picks a battery scheduler, our tool verifies its guarantee $C_{BatSched}.G$. Since $An_{BatSched}$ is not aware of thermal runway, not every scheduler meets the guarantee. As our Spin verification indicates, **FGWRR** and **FGURR** satisfy it, but **GPWRR** fails because it causes the system to reach a configuration that violates (1) with $\mathsf{TN}(0) = \mathsf{TN}(3) = 0, \mathsf{TN}(1) = 8, \mathsf{TN}(2) = 1$. Thus, our approach detects possibility of thermal runaway even though the existing analysis $An_{BatSched}$ does not.

Scalability of Contract Verification. We evaluate the scalability of our approach by comparing it to an alternative based on a unified semantic model. We focus on model checking since it is by far the most expensive component of our algorithm. The execution semantics of a unified model would, at the very least, consist of the interleaving of the two Kipke structures – $\mathcal{K}([\![\mathcal{T}_{Sched}]\!])$ and $\mathcal{K}([\![\mathcal{T}_{Therm}]\!])$. Model checking this interleaving would be intractable due to statespace explosion. In contrast, our approach is compositional and always verifies $\mathcal{K}([\![\mathcal{T}_{Sched}]\!])$ and $\mathcal{K}([\![\mathcal{T}_{Therm}]\!])$ in isolation.

We evaluated our Promela programs using a general-purpose Amazon EC2[11] virtual machine with 8 cores and 30 Gb memory. The worst-case exploration times by scheduler for the full statespace $\mathcal{K}([\![\mathcal{T}_{Sched}]\!])$ and $\mathcal{K}([\![\mathcal{T}_{Therm}]\!])$ are shown in Tab. 3 and Tab. 4, respectively. For the former we use threads with implicit harmonic periods, and for the latter we grow the battery size, fixing the output voltage requirement to $\mathsf{SerialReq} = \mathsf{ParalReq} = 3$. Although the complexity growth is exponential, $\mathcal{K}([\![\mathcal{T}_{Sched}]\!])$ is verifiable upto 6-10 threads (per CPU), and $\mathcal{K}([\![\mathcal{T}_{Therm}]\!])$ is verifiable up to batteries with 25 cells. We believe that this enables verification of realistic CPSs. We expect that other techniques, such as abstraction and symbolic model-checking, will help us to push these limits even further.

Tool Extensibility. Our tool can be extended with new verification domains. This is done by creating a new σ that defines $\mathcal{A}$ and $\mathcal{S}$ in AADL and developing a verification engine that interprets $\mathcal{T}$ and $\mathcal{R}$, which can be done separately by teams of domain experts. In addition, it is possible to incorporate new verification tools (e.g., UPPAAL instead of Spin for σ_{Sched}) as well as new analysis tools (e.g., a higher-fidelity method of calibrating K to predict thermal runaways). Finally, our approach is not fundamentally limited to AADL: any typed architecture description language can be used to create a model database.

[11]Available at aws.amazon.com/ec2.

[12] All times are in seconds. MEMLIM indicates that the verification exceeded the memory limit of 30Gb.

7. RELATED WORK

Contracts, assume-guarantee reasoning, and architectures have been used extensively to enable modular verification. In particular, in the development and verification of CPS, contracts provide an important alternative to a unified semantic model. For instance, Torngren et al. [26] use architectural viewpoints contracts as a coordination tool for designers with tools from different domains without, however, rigorously verifying application of these tools. Rajhans et al. [22] use an architectural multi-view approach where different modeling notations are captured in different views of an architectural model, using structural and semantic mappings to ensure consistency. In contrast, we concentrate not on model consistency, but on correct analysis interaction.

Sangiovanni-Vincentelli et al. [24] use contracts between components and platform-based design to combine the semantics of multiple domains. A similar approach is used in the SPEEDS project [2] to enable speculative design to support teams of distributed designers. Specifically, the authors propose the use of "rich" components where functional and non-functional aspects of the system are combined. We do not require a model that semantically unifies multiple domains, and our focus is not on the interaction between *components*. Instead we use contracts to capture the semantics of, and interaction between, the *analyses* themselves.

Assume-guarantee reasoning for control theory has been widely explored. Frehse et al. [10] develop assume-guarantee reasoning for hybrid systems based on over-approximation by simulation to enable compositional reasoning. A similar approach is taken by Girard and Pappas [11] using bisimulation, simulation, and language inclusion to develop approximate system relationships. These approaches are focused on the intersection of control theory and computer science. In contrast, our work aims at capturing a larger set of domains with extensible representations of domains.

Cofer et al. [6] present an approach to add architectural contracts to AADL components to enable compositional verification. While we share the AADL platform, we use the AADL annexes to specify contracts on the analysis and within these contracts we specify the components accessed by these analyses. The FUSED [3] project defines a meta-language approach to merge notations from multiple analysis domains to enable a syntactic integration of multiple tools. It is limited to what the syntax can express and unable to reason semantically. For instance, it would not be able to reason about when different scheduling algorithms are equivalent depending on different parameters of the taskset. In our approach we model the semantics in the analyses algorithms and are able to reason about these behavioral differences that are hidden at the syntactic level.

Our prior work on contracts[19] has been extended in a number of aspects. Previously, contracts were restricted to

a single domain – resource allocation. Contract verification was unsound and incomplete since it explored the system statespace only up to a finite depth, and the previous implementation was tied to a specific tool, Alloy, which cannot do unbounded model checking or SMT solving. In contrast, our contract language supports multiple domains, our algorithm is sound and exhaustive, and our implementation relies on extensible use of SMT solvers and model checkers.

8. CONCLUSION AND FUTURE WORK

In this paper we presented an analysis integration approach for the development of cyber-physical systems. Our approach uses novel *analysis contracts* to formally specify and automatically verify interactions between analyses from different engineering domains. Our contract language can express both static and runtime aspects of models to which analyses are applicable. We presented the syntax and formal semantics of contracts, as well as an algorithm that orders a set of analyses in a correct way and checks the applicability of each analyses via co-operative use of SMT solving and model checking. We discussed how our approach captures the semantics of a new domain in order to support analysis specific to it. Contrasting with other approaches, we show how we avoided developing a unifying semantics (which leads to an intractable verification process), focusing instead on the interactions between domain-specific analyses. We then presented the integration of the analysis contracts into the AADL/OSATE toolkit with the use of an AADL annex. Finally, we used an example to demonstrate the application of our framework to realistic models. As future work we plan to explore more scalable contract verification tools and define contracts for the contract verification tools themselves.

9. REFERENCES

[1] N. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings. Hard real-time scheduling: The deadline-monotonic approach. In *Proc. of IEEE Workshop on Real-Time Operating Systems and Software*, pages 133–137, 1991.

[2] L. Benvenuti, A. Ferrari, L. Mangeruca, E. Mazzi, R. Passerone, and C. Sofronis. A contract-based formalism for the specification of heterogeneous systems. In *Proc. of FDL*, 2008.

[3] M. Boddy, M. Michalowski, A. Schwerdfeger, H. Shackleton, and S. Vestel. The FUSED Meta-Language and Tools for Complex System Engineering. In *Proc. of AVICPS*, 2011.

[4] S. Chaki, A. Gurfinkel, S. Kong, and O. Strichman. Compositional Sequentialization of Periodic Programs. In *Proc. of VMCAI*, 2013.

[5] Y. Chen and J. W. Evans. Thermal analysis of Lithium-Ion batteries. *J. of The Electrochemical Society*, 143(9):2708–2712, Sept. 1996.

[6] D. D. Cofer, A. Gacek, S. P. Miller, M. W. Whalen, B. LaValley, and L. Sha. Compositional verification of architectural models. In *Proc. of NFM*, 2012.

[7] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proc. of TACAS*, 2008.

[8] D. De Niz and R. Rajkumar. Partitioning bin-packing algorithms for distributed real-time systems. *IJES*, 2(3), 2006.

[9] P. H. Feiler and D. P. Gluch. *Model-Based Engineering with AADL - An Introduction to the SAE Architecture Analysis and Design Language*. SEI series in software engineering. Addison-Wesley, 2012.

[10] G. Frehse, H. Zhi, and B. Krogh. Assume-guarantee reasoning for hybrid I/O-automata by over-approximation of continuous interaction. In *Proc. of CDC*, 2004.

[11] A. Girard and G. J. Pappas. Approximate bisimulation: A bridge between computer science and control theory. *Eur. J. Control*, 17(5-6), 2011.

[12] K. Hahnsang and K. G. Shin. Scheduling of battery charge, discharge, and rest. In *Proc. of RTSS*, 2009.

[13] G. J. Holzmann. The model checker spin. *IEEE Trans. Software Eng.*, 23(5), 1997.

[14] X. Jin, J. V. Deshmukh, J. Kapinski, K. Ueda, and K. Butts. Powertrain control verification benchmark. In *Proc. of HSCC 2014*, pages 253–262, New York, NY, USA, 2014. ACM.

[15] G. Karsai and J. Sztipanovits. Model-integrated development of cyber-physical systems. In *Proc. of SEUS*, 2008.

[16] G. Kim and A. Pesaran. Analysis of heat dissipation in Li-Ion cells & modules for modeling of thermal runaway. In *Proc. of 3rd International Symposium on Large Lithium-Ion Battery Technology and Application*, Long Beach, CA, 2007.

[17] H. Kim and K. Shin. On dynamic reconfiguration of a large-scale battery system. In *RTAS 2009*, pages 87–96, Apr. 2009.

[18] C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *JACM*, 20(1), 1973.

[19] M. Nam, D. de Niz, L. Wrage, and L. Sha. Resource allocation contracts for Open Analytic Runtime models. In *Proc. of EMSOFT*, 2011.

[20] OSATE 2 - An Open-Source Tool Platform for AADL v2. http://www.aadl.info.

[21] A. Pnueli. The Temporal Logic of Programs. In *Proc. of FOCS*, 1977.

[22] A. Rajhans, A. Y. Bhave, I. Ruchkin, B. Krogh, D. Garlan, A. Platzer, and B. Schmerl. Supporting heterogeneity in cyber-physical systems architectures. *IEEE Trans. Autom. Control, Special Issue on Control of CPS*, 2014. To appear.

[23] S. Saewong and R. Rajkumar. Practical voltage-scaling for fixed-priority rt-systems. In *Proc. of RTAS*, 2003.

[24] A. L. Sangiovanni-Vincentelli, W. Damm, and R. Passerone. Taming Dr. Frankenstein: Contract-based design for cyber-physical systems. *Eur. J. Control*, 18(3), 2012.

[25] J. Sztipanovits, X. Koutsoukos, G. Karsai, N. Kottenstette, P. Antsaklis, V. Gupta, B. Goodwine, J. Baras, and S. Wang. Toward a science of cyber physical system integration. *Proc. of the IEEE*, 100(1):29–44, Jan. 2012.

[26] M. Törngren, A. Qamar, M. Biehl, F. Loiret, and J. El-khoury. Integrating viewpoints in the development of mechatronic products. *Mechatronics*, 2013.