ACTIVE: A Tool for Integrating Analysis Contracts

Ivan Ruchkin[1] Dionisio De Niz[2] Sagar Chaki[2] David Garlan [1]

[1]Institute for Software Research, Carnegie Mellon University, USA, {iruchkin,garlan}@cs.cmu.edu
[2]Software Engineering Institute, Carnegie Mellon University, USA, {dionisio,chaki}@sei.cmu.edu

Abstract

In this paper we present ACTIVE a tool and framework to semantically integrate independently-developed analysis plugins in OSATE, a tool for modeling systems in the Architecture Analysis and Design Language (AADL). In the paper we analyze the problems that occur when independently-developed analysis pluging are executed on an AADL model and how these problems can lead to invalid analysis results. We show how our framework model plugin interactions in a formal way in order to enable automatic verification. These interactions are captured in an analysis contract that describes inputs, outputs, assumptions, and guarantees. The input and outputs in the contract allow us to determine the correct order in which plugins must execute. The assumptions and guarantees, on the other hand, capture the conditions that must be valid to execute a plugin and the one that are expected to be valid afterwards. The ACTIVE framework allows the inclusion of any generic verification tool (e.g. model checkers) to verify these conditions. To coordinate all these activities uses two components: the ACTIVE EXECUTER and the ACTIVE VERIFIER. The ACTIVE EXECUTER executes the plugins in the required order calling the ACTIVE VERIFIER every time assumption and guarantees need to be verified. The ACTIVE VERIFIER, in turn identifies and executes the verification tool that needs to be invoked based on the target formula. Together, they ensure that the plugins are always executed in the correct order and under the correct conditions guaranteeing a sound verification process. To the best of our knowledge ACTIVE is the first extensible framework able to integrate independently-developed analysis plugins guaranteeing their correct interaction and execution.

Keywords analysis contracts, cyber-physical systems, model checking, virtual integration

1. Introduction

The development of Cyber-Physical Systems (CPS) relies on analysis tools that use their own specialized abstractions. These abstractions, however, interact with each other. Neglecting these interactions leads to incorrect analysis results and design choices. For instance, selecting an otherwise valid scheduling policy may violate the assumptions of a functional correctness analysis (e.g., model checking) and lead to a deadlock-prone system being declared safe. Alternatively, such a scheduling policy could violate the assumptions of a processor frequency scaling analysis and lead to a non-schedulable allocation of tasks. Similarly, modifying the algorithm of a controller may alter its execution time and periodicity, thereby affecting the schedulability of the system [3].

Verification tools are particularly sensitive to their underlying domain abstractions, and incorrect application of these abstractions leads to invalid results. For instance, the original schedulability equations for Rate-Monotonic Scheduling (RMS) [6] use task abstractions that both restrict these tasks to be independent of one another and to forbid them to pause their computation. The application of this abstraction to tasks that do not honor these restrictions leads to incorrect schedulability results via RMS.

The process of virtual integration aims at addressing the issue of dependent abstractions by providing methods to resolve dependencies and conflicts among tools and models used in CPS engineering [7]. In our recent work we developed an approach for virtual integration based on the specification and verification of *contracts* between analysis tools [8]. Our approach allows us to describe and verify the interactions between analyses from different scientific domains.

Our approach relies on specification of a *verification domain* – a set of symbols and their interpretation that enables us to express the contracts of analyses belonging to that domain in a precise manner. Examples of domains are “scheduling” and “battery,” which are concerned with analyses of properties of real-time threads (e.g., schedulability), and batteries (e.g., thermal runaway), respectively. A verification domain serves as a signature to a mixed-logic language in which dependencies, assumptions, and guarantees of each analysis from this domain are specified in the form of a contract. These contracts are then algorithmically evaluated against a particular system architecture to verify if the system satisfies the assumptions of the analyses, and if the contracts of the analyses are compatible with one another.

The theoretical constructs from our prior work are helpful, but, unfortunately, insufficient to support practical in-

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE **14 OCT 2014**	2. REPORT TYPE **N/A**	3. DATES COVERED **-**

4. TITLE AND SUBTITLE **ACTIVE: A Tool for Integrating Analysis Contracts**	5a. CONTRACT NUMBER
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) **Dionisio de Niz Ivan Ruchkin; Sagar Chaki; David Garlan**	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release, distribution unlimited

13. SUPPLEMENTARY NOTES
The original document contains color images.

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**SAR**	**9**	

tegration of analysis tools. CPS analysis and verification tools have many peculiarities in terms of how they are initialized and executed. For example, many user-facing tools may not provide adequate ways to execute them from a virtual integration tool. At the same time, specifications of verification domains and contracts need to be, on the one hand, reused to avoid unnecessary duplication, and, on the other hand, may need to be adjusted correctly to each model's context. A missing part is a virtual platform that can manage tool interactions at a practical level. One key requirement for such a platform is extensibility: the aforementioned challenges of adding new tools and verification domains have to be minimized in order to achieve the theoretical promises of analysis integration.

In this paper, we present our tool ACTIVE [1] (Analysis Contract Integration Verifier), which implements the described analysis integration approach. Developed on top of the OSATE2 [2] toolkit, ACTIVE uses the AADL [4] architectural description language because AADL offers a convenient way to represent the structural, design-time aspect of the system. In particular, the syntactic core of AADL is tailored for embedded systems, and AADL annexes allow us to add a specialized sublanguage for the needs of a specific analysis. In addition to AADL, in order to verify properties that vary during runtime ACTIVE uses a mixed specification language that includes Linear Temporal Logic [9] along with a model checker to perform the verification.

Since our goal is to develop an extensible platform for virtual integration of CPS analyses, ACTIVE already incorporates several analysis tools (e.g., a bin packing algorithm for thread-to-processor allocation) and contract verifiers (like SMT and Spin). To this end, ACTIVE includes (i) a language to describe analysis contracts, (ii) a mechanism to execute analyses in the correct order, and (iii) a contract verification engine to verify if the analysis contracts hold. All three parts are extensible with new verification domains (e.g., controller simulation) and types of verification (e.g., probabilistic model checking).

The rest of the paper is organized as follows. In Section 2 we break down and discusses the problem of managing analysis tool interactions. Section 3 focuses on the representation, while Section 4 and 5 demonstrate our solutions to behavioral challenges of using multiple analysis and verification tools. Section 6 concludes this paper.

2. Managing Interaction of CPS Analysis Plugins

AADL provides a description language to capture the main structures and components of a CPS (e.g., threads and processors), and the sub-systems they belong to, along with the connections between them. Components are annotated with properties from various engineering domains. Examples of properties include thread period, processor frequency, and scheduling policy (all from the real-time scheduling domain), and required battery voltage (from the thermal analysis domain). The system designer defines component types along with their subcomponents, interconnections, and properties to create what is known as the *AADL declarative model* of a system. This model is then transformed into an *AADL instance model* – an XML-based representation of the actual system, rather than component types.

In OSATE, analysis algorithms are implemented as plugins that have access to both the declarative and the instance AADL models. AADL allows the augmentation of these models with tool-specific descriptions known as annexes that define sublanguages embedded in AADL descriptions. For example, the error model plugin [4] relies on an annex to specify error sources and propagations. It reads components and properties from the instance model and error annex specifications from the declarative model to produce, for example, a fault impact report.

While OSATE supports the rapid development of analysis plugins, it does not support controlling undesirable interactions between different plugins in order to prevent incorrect results. In particular, OSATE provides extension points to add annexes and invocation actions (via toolbar buttons or menu items). Once a plugin is invoked, it is given access to AADL models to carry out the desired operations and return the control back to the user. Each analysis plugin thus tends to focus on a specific technical concern, accessing components, properties, and annex clauses relevant to that concern, but ignoring the effects of other plugins.

The lack of support to capture and control the effects that plugins have on each other can lead to incorrect executions of these plugins. For instance, a plugin that modifies the thread-to-processor allocation might invalidate the result of any prior schedulability plugin applied to the affected threads and processors. Similarly, if a plugin depends on the scheduling policy of a processor (say Rate-Monotonic Scheduling) and another plugin modifies this policy (to, say, Earliest-Deadline First) the results could be invalidated. Worst of all, more often than not, these errors go unnoticed given the lack of an infrastructure to capture and verify these plugin interactions. Unfortunately, OSATE currently lacks ways to manage the co-operative execution of analyses, let alone verify their assumptions formally.

To address the issue of analysis interaction systematically, *an integration tool* needs to manage analysis plugins at the level of the abstractions that these analyses use. We identify three parts of this problem: (i) describing plugin interactions with enough details that not only captures real conflicts but also avoids signaling false ones; (ii) executing plugins in a right sequence and at a right time; and (iii) verifying applicability assumptions of analysis plugins. All three parts need to be addressed in an *extensible way* so as to not lose the benefits of the OSATE design. We go deeper into each part of the problem below.

2.1 Representing Plugin Interactions

In order to enable the analysis of plugin interactions we must model these interactions. In particular, first, we need to represent data flows between the plugins and the AADL models. In other words, we need to identify the parts of the model that the different plugins change. This dataflow description allow us to prevent the incorrect order of plugin

[1] Can be downloaded at [url provided in the final version].

[2] `https://wiki.sei.cmu.edu/aadl/index.php/Osate_2`

execution where the output from one plugin is invalidated by the output from another plugin executed afterwards.

The second aspect that needs to be represented is the assumptions a plugin makes about the model under analysis. For example, there are several implementations of schedulability tests that rely on rate-monotonic assumptions without stating them explicitly. More importantly, plugin assumptions may be related to the system behavior rather than its structure. Thus predicates over only the structural models in AADL are inadequate for expressing such assumptions. For example, one plugin may assume that a task may only be preempted by others with shorter deadline than its own. This assumption is trivially satisfied if we used Deadline Monotonic Scheduling. However, it could also be satisfied with other scheduling policies under certain specific system configurations. This can only be discovered with enough information about the behavior of the system with the different policies.

Finally, the way an analysis is identified in a specification must reflect the operations performed by the analysis on a model precisely. Since many plugins come with several related operations; for example, a resource allocation plugin provides three operations: allocation of threads to processors (what we further call bin packing), a utilization-based schedulability test, and a priority inversion test. Just mentioning the name of the resource allocation plugin does not differentiate the operations properly.

One of the biggest challenges to represent plugin interactions come from the extensibility requirement. First of all, we must be able to change the plugin interaction specification independently from the plugin implementation. This is because the AADL semantics allows different plugins to have slightly different interpretations of the properties. For instance, we have seen AADL models where threads are assumed to be periodic by default while other models explicitly require the use of the `Periodic` value in the `Dispatch_Protocol` property, and yet others use the `Hybrid` value for the same property.

Another important aspect of the extensibility is the support of new domains that may introduce new types of components, properties, or even behavioral specification constructs. For example, to introduce analyses for multi-cell reconfigurable batteries, one would have to introduce a new device type, with new AADL properties (such as size), and runtime properties to capture the cell connectivity. These changes must affect only AADL models, and not the plugin integration tool.

2.2 Correct Plugin Execution

To ensure the correct execution of a plugin it is necessary to respect its data dependencies and ensure that its assumptions are never violated as different plugins are executed. In practice, a *correct plugin execution* implies the following steps: (i) before the plugin is called we need to ensure that all the other plugins have finished their work and committed their changes to the AADL model; (ii) the plugin assumptions must be validated on the model on which the plugin will be run; (iv) if an analysis plugin ends with an error, the sequence of execution has to be stopped.

Many OSATE plugin are made with human user interaction in mind: they expect to be run from toolbars and menus. Since OSATE discourages plugin interaction, it is challenging to invoke plugins programmatically. Once a plugin has been invoked, monitoring its progress is a challenge as well: many plugins use tools external to OSATE, without any feedback. The limited feedback mechanisms in OSATE were also designed with a human user in mind, making it difficult to monitor analysis execution and determine the time when it is safe to start the next analysis.

Like the specification, correct plugin execution needs to be achieved with minimal changes to existing plugins and the OSATE tool. Plugins still need to run individually on user's command, and major changes to their control flow are not acceptable. OSATE cannot be profoundly modified either. In particular, it is important to leave the option open to run plugins without the proper integration in cases when specifications have not been completed, or they do not hold.

2.3 Extensible Assumption Verification

An analysis plugin integration tool needs to use state of the art verifiers to ensure that analyses are only used when they are known to produce correct results. Since the analysis plugin integration problem manifests itself in multiple domains that contribute to CPS, verifiers need to be tailored to domain-specific abstractions and applied in their corresponding contexts. For example, a dynamic model of a thread scheduler cannot be used to verify assumptions about a flight controller behavior. In addition, given the existence of specifications involving static and dynamic properties of the system it is necessary to enable the verification of both types of properties in a scalable way.

AADL models are organized hierarchically: a set of threads may be composed in a thread group, which in turn contributes to a software subsystem. A software subsystem is part of a computational subsystem, which also includes processors and memory devices. Finally, the computational subsystem is part of the whole system, which also includes physical devices (such as rotors), and properties (such as mass). The hierarchy is left up to the engineer, allowing multiple ways to describe the same system.

Unfortunately, many verifiers rely on their own system decomposition, which may not agree with AADL. Tools for timed automata verification, such as UPPAAL [5], use the refinement relation rather than an arbitrary composition. Hence, our challenge is to create mechanisms that uses (yet unknown) verifiers for a custom-built AADL model at a proper level of hierarchy. Domain-specific verifiers, for instance, need to access their own abstractions without being aware of other components.

The three following sections present our solutions in ACTIVE to the aforementioned problems. We view our solutions from two positions: functionality – achieving their goal correctly – and extensibility – allowing addition of new elements, such as analyses and verifiers, to the tool.

3. Contract Language as AADL Annex

The location of the specification of the analysis interactions is a decision that affects the extensibility of our framework. We evaluated three potential locations: (i) inside the plugin itself; (ii) in the OSATE tool; or (iii) in the AADL model. Encapsulating the specification inside the plugin has the benefit that the description always travels with the plugin. On the other hand, storing the specification in a central database of specifications inside the OSATE tool facilitates collaboration and reuse of contracts. Unfortunately, both of this options diminishes extensibility given that the exact specification of the interactions may change depending on the project and the plugins used in it, as discussed in Section 2. That is, semantic variations in the interpretation of AADL in a particular model need to accounted for without changing the plugin.

In ACTIVE we use an AADL annex (like many analysis plugins do) to represent analysis dependencies and assumptions. In this case, an annex instance is attached to a declarative AADL model and can be used whenever this model, or any derived instance model, is used. We refer to our analysis interaction specification as an *analysis contract*. An analysis contract has the following parts:

- **Name**: names the analysis contract as well as the analysis plugin wrapper to be called when an analysis is invoked (see Section 4 for more information on analysis wrappers).
- **Input**: a comma-separated list of elements: AADL component types (e.g., `thread`) and property names (e.g., `thread.Period`). The property names have to be prefixed with a component type to identify the dependency more precisely. By including a component type in its input, an analysis plugin declares that it accesses the set of components; and by including a property, a plugin declares that it reads the property values. If a property is included as part of the input or output, the related component is assumed to be included as well.
- **Output**: a list of same type of elements as that of the input. The only difference is semantic: the analysis declares that it changes the set of components or the values of a property. Although the specification of inputs and outputs is in terms of AADL types, the changes are meant to be done to the instance model.
- **Assumes**: a set of assumptions that have to hold for an analysis to be applicable. Each assumption is a logical formula, as explained below. Given that sometimes we may not have a complete definition of when an analysis is applicable, we need to make sure that it is applicable at least under the assumptions included here. If an analysis is always applicable, this part can be omitted.
- **Guarantees**: a set of formulas that must be true on the model after the analysis has been executed. These formulas are syntactically equivalent to the assumption formulas. Guarantees can fix conditions that are expected to be true after an analysis plugin runs. These conditions can then be used to satisfy the assumptions of other contracts. In such a case, the assumption of these contracts do not need to be reverified. However, if the guarantee cannot be met, the assumptions of the contracts that depend on it must be reverified.

Assumes and guarantees contract formulas have the following syntax in ACTIVE:

```
(
  ['forall' | 'exists']
  (<Var>:<Type> ',')+
  (
    '|' <PredicateExpression>
  )?
  ':'
)?
<LTLExpression>
```

Here, `<Var>` introduces a quantified variable name of an AADL component having type `Type`. Variables are quantified over `<PredicateExpression>`, which is a logical predicate over the AADL's model components and properties with the usual logical operators `and`, `or`, and `not`. The `<LTLExpression>` encodes a domain-specific behavioral property using a combination of logical operators above and the LTL modalities Globally `G`, Eventually `F`, and Until `U`. However, if necessary, `<LTLExpression>` can be limited to predicate logic.

The operator "`:`" is implicative when used in a `forall` formula (where it denotes that "all variable valuations that satisfy condition `<PredicateExpression>` should also satisfy `<LTLExpression>`") and conjunctive when used in a `exists` formula (where it denotes that "all variable valuations that satisfy condition `<PredicateExpression>` should also satisfy `<LTLExpression>`"). We do not use Quantified LTL [9] because it prevents a cleaner split of formulas between general-purpose SMT solvers and domain-specific verifiers (see details in Section 5), and brings in unnecessary complexity.

Let us illustrate the application of contracts with examples from the scheduling domain. Figure 1 shows a contract for a processor frequency scaling analysis. The goal of this analysis is to minimize the processor frequency to limit energy expenditure of the system. The analysis reads threads, processors, thread deadlines, and thread bindings (allocations) to processors. The output is the CPU frequency. An implicit but important assumption of this analysis is that the tasks in the system run under the deadline monotonic scheduling policy. This is captured in a formula stating that "every pair of distinct threads allocated on the same processor should behave as if scheduled by a deadline-monotonic policy." The first part of the formula (before the colon) indicates the condition for which all possible pairs of threads should be evaluated. The second part (after the colon) is an LTL expression that features a domain-specific predicate `CanPreempt`, which is true in any state during runtime if and only if `t_1` is executing but `t_2` is ready to execute, but not executing.

Another example of an analysis is verification of safe concurrency based on the tool LLREK [1]: the tool takes

```
-- Frequency scaling analysis
-- Adjusts processor frequency based on thread bindings
-- Assumes that threads on the same processor
--      behave equivalently to DMS
name FreqScalingAnalysis
input
    thread, thread.Actual_Processor_Binding,
    thread.Deadline, processor

output
    processor.Current_Frequency

assumes
    forall t1:thread, t2:thread | t1 != t2 &&
    t1.Actual_Processor_Binding[0] = t2.Actual_Processor_Binding[0]:
    G(CanPreempt(t1, t2) => t1.Deadline < t2.Deadline)
```

Figure 1: Contract for frequency scaling analysis.

```
-- Llrek thread model checking
-- Checks concurrency violations: deadlocks and race conditions
-- Assumes that preemption does not alternate
name LlrekAnalysis
input
    thread, thread.Period, thread.Deadline,
    thread.Source_Text, thread.Priority,
    thread.Compute_Execution_Time, thread.Priority

output
    system.Is_Thread_Safe

assumes
    forall x1:thread, x2: thread | x1 != x2 :
    G (CanPreempt(x1, x2) => (G !CanPreempt(x2, x1)))
```

Figure 2: Contract for the LLREK analysis.

```
annex contract {**
    use contracts::SecurityAnalysis,
    contracts::BinPackingAnalysis,
    contracts::FreqScalingAnalysis,
    contracts::ThermalRunawayAnalysis,
    contracts::BatterySchedulingAnalysis
**};
```

Figure 3: Annex subclause specifying which analyses to use.

source code of each thread annotated with safety assertions and determines whether the assertions are met. A contract for the LLREK analysis is shown in Figure 2. The analysis reads a number of thread properties and outputs whether the system was found to be safe w.r.t. its annotated assertions. LLREK assumes fixed-priority scheduling, i.e., thread pre-emption is acyclic. In other words, if thread `x1` preempts thread `x2`, then `x2` never preempts `x1`.

To improve convenience and reuse of contracts, we separate the definition of contracts (as in Figures 1–2), which we call a library of contracts, from the application of these contracts (shown in Figure 3) to a system, which we call a usage subclause. Usage subclauses enable the association of analyses to models. In this way a user can control applicability of analyses at a macro-level reusing the same contracts across different models.

Our approach explained in [8] relies not only on the specification of contracts (where the mapping between conceptual and practical aspects is more straightforward), but also on verification domains that define the formal underpinnings for both the specification and the analysis of the contracts within a verification tool (e.g., SPIN). Formally, a verification domain σ is comprised of domain atoms $\mathcal{A}$, static functions $\mathcal{S}$, runtime functions $\mathcal{R}$, execution semantics $\mathcal{T}$, and domain interpretation for atoms and static functions $\llbracket \ldots \rrbracket_\sigma$. These elements are augmented by an architectural model that provides the interpretation $\llbracket \ldots \rrbracket_{\mathsf{M}}$. The existence of verification domain with a semantics defined within a verification tool that automatically explores its behavior guarantees correctness of our analysis contracts approach.

In ACTIVE, verification domains are not specified in one place, but are comprised of various elements of AADL and contract annexes. For some atoms $a \in \mathcal{A}$, $\llbracket a \rrbracket_\sigma$ is provided by OSATE. For example, integers, booleans, and reals are standard types in AADL. Other elements of $\mathcal{A}$ are specified by the $\llbracket \ldots \rrbracket_{\mathsf{M}}$, e.g., threads, processes, memory elements, processors, systems, and other sets of components.

Static functions $\mathcal{S}$ map directly to AADL's properties, some of which are standard and some of which are defined by users in a declarative model. Only static functions can be used in `<PredicateExpression>` so that the semantics of `<PredicateExpression>` could be fully constructed based on the AADL instance model values. Stardard types have a default value, which contributes to $\llbracket \mathcal{S} \rrbracket_\sigma$. For the most part, however, interpretation of static functions comes from $\llbracket \mathcal{S} \rrbracket_{\mathsf{M}}$ in the form of values that properties have in a particular AADL instance model.

Runtime functions $\mathcal{R}$ are domain-specific, e.g., `CanPreempt`. Their interpretation comes from a domain-specific verifier and, as far as AADL models are concerned, these functions do not exist. Finally, the execution semantics $\mathcal{T}$ is defined by a combination of static function specified by the model (e.g., thread periods and deadlines) and verifier-specific runtime behavior (e.g., how the state of system changes when a new thread arrives). Thus, all formal elements of σ are covered in ACTIVE, to which the formal conclusions of correctness can now be transferred.

4. ACTIVE EXECUTER

Ensuring that a set of analysis plugins are executed correctly requires more than the specification and verification of individual analysis contracts. Specifically, it requires coordination and proper sequencing of the execution of these analysis and their verifiers. This section describes ACTIVE EXECUTER– a plugin execution controller. The purpose of this controller is, on the one hand, to interact with the user and, on the other hand, to coordinate the execution of analyses. The scheme of ACTIVE EXECUTER is shown in Figure 4.

From the user interaction point of view, the ACTIVE EXECUTER identifies dependencies between analyses, builds a dependency graph, and presents it to the user to allow him to select an analysis to run. When invoked on an instance model, the ACTIVE EXECUTER parses all usage subclauses that are in the scope of the system, retrieving the corresponding contracts from annex libraries, and creating the dependency graph. In this graph, each vertex represents a contract with its corresponding analysis plugin. Edges in this graph represent input-output dependences between two contracts.

In this context we say that an analysis plugin depends on another if the former reads a property or a component set

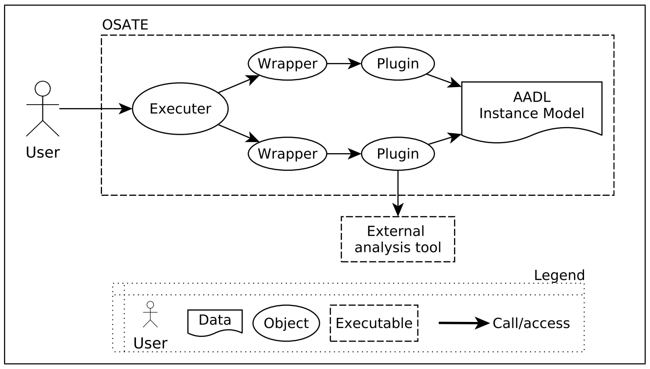

Figure 4: Operation of ACTIVE EXECUTER.

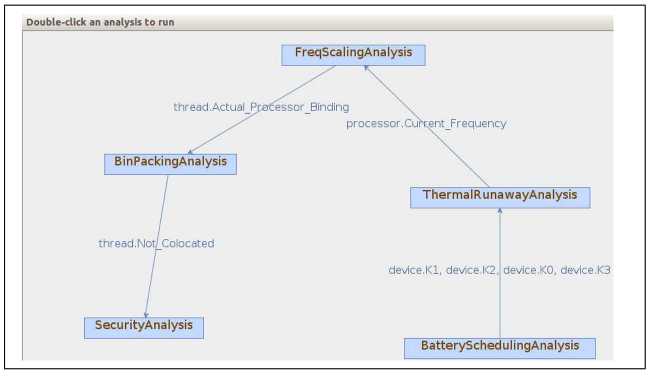
Double-click an analysis to run
FreqScalingAnalysis
thread.Actual_Processor_Binding
processor.Current_Frequency
BinPackingAnalysis
ThermalRunawayAnalysis
thread.Not_Colocated
device.K1, device.K2, device.K0, device.K3
SecurityAnalysis
BatterySchedulingAnalysis

Figure 5: Analysis selection window in ACTIVE.

that the latter modifies. Since inputs and outputs are specified in terms of component types and properties, any plugin that writes a component type is dependent on any analysis that reads a property of this type given that, formally speaking, the former changes the domain of the function. The opposite is not true: changing a property does not constitute a change in component.

The dependency graph is used by the user to select an analysis to run. A screenshot of a ACTIVE's selection window is shown in Figure 5: the rectangles represent analysis plugins and the arrows show the dependency relationships. Once a selection is made, the ACTIVE EXECUTER finds a correct sequence of analyses leading up to the selected one (the formal details can be found in [8]). For example, for the analyses in Figure 5, the execution of the frequency scaling plugin would first require the execution of the security plugin followed by the binpacking plugin.

Currently, ACTIVE only supports acyclic dependencies. In practice a cyclic dependency can be a symptom of incorrect contract specifications. However, when this is not the case and the cycle is not an error it may be possible to execute each analysis in the cycle repeatedly till convergence (i.e., executing any analysis in the cycle does not change the system further, or in other words, a fixed point is reached). We leave this investigation as future research.

Invoking the right analysis plugin is not as trivial as the theory assumes. If ACTIVE's code were to call analysis plugins directly, ACTIVE would have a direct dependency on a concrete set of plugins and would not be deployable separately from these plugins. Even further, plugins developed for human user have external access points such as toolbar

```
<!-- For integration with contracts: used to run a Binpack action -->
<extension
      id="binpack.command"
      point="org.eclipse.ui.commands">

     <category
         name="Resource management commands"
         id="org.osate.analysis.resource.management.commands">
     </category>

     <command
         categoryId="org.osate.analysis.resource.management.commands"
         id="org.osate.analysis.resource.management.commands.Binpack"
         description="Execute the binpacking action through the command framework"
         name="binpackcommand">
     </command>
</extension>
```

Figure 6: A plugin wrapper's command interface for binpacking analysis.

buttons and menus, that cannot be called programmatically from another OSATE plugin.

To overcome these limitations and use analysis plugins without substantial changes to their external interface, we developed *analysis plugin wrappers* – a method to execute analysis plugins using the Eclipse Command Framework [3]. A wrapper creates a command interface around the user action. A command, unlike an action, can be called programmatically, and results in a call of an associated action. Each plugin wrapper thus consists of a command interface – which is an addition to the plugin configuration exemplified in Figure 6 – and a direct command invocation that can be exercised by the ACTIVE EXECUTER.

Another factor that affects the correct execution of plugins is understanding when it is safe to start the execution of the next plugin in a dependency chain. Therefore, the execution of every plugin needs to be monitored. Unfortunately, there is no direct way to do so in the original implementation of OSATE. Our ACTIVE EXECUTER relies on the plugin wrappers to perform this monitoring. Specifically, when a wrapper starts a command associated with an action, the progress of this command is tracked using the identity of the associated action. Then, once the wrapper reports that the plugin has finished, the next plugin is safely started.

To check whether the assumptions of a plugin hold before its execution, or if its guarantees hold after its execution, the ACTIVE EXECUTER calls the ACTIVE VERIFIER– another key component of ACTIVE– passing over a contract and a formula to verify before and after the execution of the plugin. We discuss the details of this process in the next section.

5. ACTIVE VERIFIER

The verification of analysis assumptions is the most complex aspect of ACTIVE given that it requires the combination of abstractions coming from diverse scientific fields into a common logic. Its goal is to take a contract formula and verify it against the current AADL instance model. This verification is, however, not limited to the AADL model, which abstracts away most of the behavioral system dynamics given its architectural nature. A further complication comes from assumption formulas with abstractions and properties from different verification domains, which

[3] http://wiki.eclipse.org/Platform_Command_Framework

manifest at different levels of the hierarchy in an AADL model. Last but not least, the addition of new verification tools and models, which is at the core of virtual integration, needs to be facilitated minimizing changes to the ACTIVE or verifier structure.

To address the challenges of multi-domain verification, we created the component ACTIVE VERIFIER. The first step in the verifier's operation is to deconstruct the contract formula to pieces that can be processed by an individual verifier. The variable quantification and the `<PredicateExpression>` can be processed with a general-purpose SMT solver since, as we showed in Section 3, all atoms and operators of `<PredicateExpression>` are determined from the AADL models, thus rendering quantified `<PredicateExpression>` amenable to an efficient validity check. The practical purpose of `<PredicateExpression>` is to identify the part of a model that the formula should apply to, thus providing a convenient access to the hierarchy and bypassing irrelevant parts. This way, the frequency scaling plugin,for instance, can indicate that it targets only pairs of threads running on the same processor (Figure 1), even if the processor is located in a hardware subsystem, separated from threads by several AADL hierarchy levels.

The ACTIVE VERIFIER reduces the search for variable valuations that satisfy `<PredicateExpression>` (as well as a validity check for `<LTLExpression>` without domain-specific atoms or LTL operators) to a SMT formula φ generated from the AADL components and properties with an added assertion of negated `<PredicateExpression>`. To construct φ, ACTIVE VERIFIER only explores the subset of the AADL model that includes the components and properties mentioned in the contract formula. It then checks the satisfiability of φ using an off-the-shelf SMT solver (currently Z3 [2]). If φ is satisfiable, the solution is recoded and blocked for the next run. If φ is UNSAT, the search is stopped, and the verification moves to process the `<LTLExpression>`. Thus, by using "blocking clauses" incrementally, ACTIVE VERIFIER generates all solutions of φ.

The verification of `<LTLExpression>`, however, cannot rely on a general-purpose SMT solver since `<LTLExpression>` may contain domain-specific runtime functions like `CanPreempt` and modal LTL operators. Thus, an `<LTLExpression>` needs to be matched with a domain-specific verifier, based on a verifiers' fitness. Specifically, we say that a verifier *matches a formula* if and only if this verifier can give an interpretation to every atom (such as set or function) and every operator in the formula. Typically matching is somewhat known to engineers familiar with particular verifiers, but not explicitly documented in a machine-readable form. Without a proper representation of matching, verifier selection decisions would risk an error of running a verifier on an inappropriate formula producing invalid results. For example, a non-preemptive scheduler model would report non-schedulability on many systems where a preemptive scheduler would report schedulability.

The problem of matching verifiers to contract formulas led us to develop an abstraction of *verification engines* – an abstraction to simplify the access to verifiers and determine formula matching. Each verifier is augmented with a verification engine that governs the application and execution of the verifier through the following functions:

- The verifier's initialization and parameter selection. For instance, to run the Spin verifier, used in the scheduling domain, it is necessary to translate AADL properties into Promela instructions to complete the template of a Promela program for preemptive schedulers.
- The declaration of atoms and operators that can be interpreted by the managed verifier. For example, in the scheduling Spin verification engine reports `CanPreempt` as an atom and LTL operators `G`, `F`, and `U`.
- The declaration of the AADL model parts that are required to achieve full a semantic interpretation of $\mathcal{T}$. For example, to generate traces of a thread scheduler, a Spin program needs to read thread periods from an AADL model.
- The interpretation of the results from the verifier. While theoretically a verifier could have only two answers $\top$ and $\bot$, in practice other options are possible: a verifier may detect a syntax error or run out of memory. These results do not necessarily constitute a violation of contract, and verification engines report those as "verification not possible," thus making user interaction more transparent.

The verification engine abstraction allows ACTIVE to handle diverse verifiers using a common interface.

The pseudocode of the end-to-end algorithm of verifying a contract formula is shown in Listing 7. In this algorithm the ACTIVE EXECUTER, first calls the function VERIFY, with a contract formula and a model. If the SMT solver can fully handle the formula (i.e., there are no domain-specific atoms or operators), the ACTIVE VERIFIER takes a shortcut and delegates the verification exclusively to the SMT. If this shortcut is not possible, the ACTIVE VERIFIER searches for a matching verifier (function MATCH) and runs the selected verifier on every valuation of the variables (function RUN).

Up to this point we have discussed our approach to coordinate the application of a set of verifiers. However, another important problem that we need to address to simplify the addition of new verifiers is the access to the parts of the model referenced by contract formulas. In particular, without a generic model access approach it would be necessary to write custom code for each verifier to access parts of the model located at different levels of its hierarchy. To simplify data collection for verifiers, we introduced the *shared-data interface* that helps to decouple domain-specific verifiers from the hierarchy of the AADL model. The data interface provides SMT and domain-specific verifiers with SQL-based access to the AADL instance model data. All components are given unique identifiers and stored in separate tables. Each AADL property is represented with a table

```
function VERIFY(Formula $f$, Model $m$)
    if SMT.$canSolve$(f) then
        return ¬ SMT.$isSat(\neg f)$
    $v \leftarrow$ MATCH$(f, m)$
    if $v \neq$ null then
        return RUN$(f, v)$
    else
        return error
function MATCH(Formula $f$, Model $m$)
    for all VerificationEngine $v$ do
        if $v.canInterpretAtoms(f.atoms)$
          $\wedge v.canInterpretOperators(f.operators)$
          $\wedge v.hasFullInterpretation(m)$ then
            return $v$
    return null
function RUN(Formula $f$, VerificationEngine $v$)
    if <Var> $\in f$ then
        $varEvals \leftarrow$
        SMT($f$.<PredicateExpression>, $m$)
        if $f.quan = exists$ then
            $res =$ false
            for all $ve \in varEvals$ do
                $res \leftarrow res \vee$
                $v.verify$ ($f$.<LTLExpression>, $m$)
        else if $f.quan = forall$ then
            $res =$ true
            for all $ve \in varEvals$ do
                $res \leftarrow res \wedge$
                $v.verify$ ($f$.<LTLExpression>, $m$)
        return $res$
    else if
      then
```

Figure 7: Algorithm of ACTIVE VERIFIER

of the same name that lists the values in a format appropriate for the property type, as well as the owner component of this property. This way, all components and properties are easily accessible to verifiers without the need to traverse levels of hierarchy in AADL. The downside of the shared-data interface is that it does not support composition of data types of arbitrary depth (e.g., a sequence of records, each having a field that is a set of other records). However, we are yet to see a plugin that uses more than three levels of recursion. We use a MySQL database to implement the interface.

6. Conclusion

In this paper we presented ACTIVE, a tool for addressing the problem of AADL analysis plugin interactions. In particular, we discussed how analysis plugin integration errors pose the risk of invalidating analysis results without user's knowledge. Solving this problem entails, first of all, representing the analysis interactions in a formal way to enable automatic reasoning. The specification of these interactions, in the form of inputs, outputs, assumptions, and guarantees, also allow us to determine the correct order in which plugins must execute. Finally, the assumptions and guarantees need to be verified using a potentially wide variety of verification tools. The representation of the analysis interactions with an AADL annex-based contract language allows the ACTIVE EXECUTER to manage the proper startup and monitoring of the analysis plugins, making the appropriate calls to the ACTIVE VERIFIER which in turn invokes a general-purpose SMT solvers (e.g., Z3) and domain-specific model checkers (e.g., Spin) for in-depth behavioral verification. These major ACTIVE components were designed to be extensible to accommodate new verification domains, analysis plugins, and domain-specific verifiers. To the best of our knowledge ACTIVE is the first extensible framework able to integrate analysis plugins guaranteeing their correct interaction and execution.

Acknowledgments

We thank Julien Delange, Peter Feiler, Lutz Wrage, and Joseph Siebel of the Software Engineering Institute for their help in implementing ACTIVE.

References

[1] Sagar Chaki, Arie Gurfinkel, and Ofer Strichman. Efficient Verification of Periodic Programs using Sequential Consistency and Snapshots. In *Proceedings of the 14th International Conference on Formal Methods in Computer-Aided Design (FMCAD '14)*, Lausanne, Switzerland, October 2014. IEEE Computer Society.

[2] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '08)*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340, Budapest, Hungary, March–April 2008. Springer-Verlag.

[3] Patricia Derler, Edward A. Lee, Stavros Tripakis, and Martin TÃűrngren. Cyber-physical system design contracts. In *Proceedings of the ACM/IEEE 4th International Conference on Cyber-Physical Systems*, ICCPS '13, pages 109–118, New York, NY, USA, 2013. ACM.

[4] Peter H. Feiler and David P. Gluch. *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language*. Addison-Wesley Professional, Upper Saddle River, NJ, 1 edition edition, October 2012.

[5] Kim G. \ Larsen, Paul Pettersson, and Wang Yi. Model-checking for real-time systems. In *Proc.\ of Fundamentals of Computation Theory*, Lecture Notes in Computer Science, pages 62–88, August 1995.

[6] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, January 1973.

[7] Panagiotis Manolios and Vasilis Papavasileiou. Virtual integration of cyber-physical systems by verification. In

Proceedings of AVICPS 2010, 2010.

[8] Ivan Ruchkin, Dionisio De Niz, Sagar Chaki, and David Garlan. Contract-based integration of cyber-physical analyses. In *Proceedings of the twelfth ACM international conference on Embedded software*, New Delhi, India, October 2014.

[9] Aravinda Prasad Sistla. *Theoretical Issues in the Design and Verification of Distributed Systems*. PhD thesis, Harvard University, Cambridge, MA, USA, 1983. AAI8403047.