An AADL Contract Language Supporting Integrated Model- and Code-Level Verification

JOHN HATCLIFF, Kansas State University, Department of Computing and Information, USA
DANIELLE STEWART, Adventium Labs, USA
JASON BELT, Kansas State University, Department of Computing and Information, USA
ROBBY, Kansas State University, Department of Computing and Information, USA
AUGUST SCHWERDFEGER, Adventium Labs, USA

Model-based systems engineering approaches support the early adoption of a model – a collection of abstractions – of the system under development. The system model can be augmented with key properties of the system including formal specifications of system behavior that codify portions of system and unit-level requirements. There are obvious gaps between the model with formally specified behavior and the deployed system. Previous work on component contract languages has shown how behavior can be specified in models defined using the Architecture Analysis and Design Language (AADL) – a SAE International standard (AS5506C). That work demonstrated the effectiveness of model-level formal methods specification and verification but did not provide a strong and direct connection to system implementations developed using conventional programming languages. In particular, there was no refinement of model-level contracts to programming language-level contracts nor a framework for formally verifying that program code conforms to model-level behavioral specifications.

To address these gaps and to enable the practical application of model-contract languages for verification of deployed high-integrity systems, this paper describes the design of the GUMBO AADL contract language that integrates and extends key concepts from earlier contract languages. The GUMBO contract language (GCL) is closely aligned to a formal semantics of the AADL run-time framework, which provides a platform- and language-independent specification of AADL semantics. We have enhanced the HAMR AADL code generation framework to translate model-level contracts to programming language-level contracts in the Slang high-integrity language. We demonstrate how the Logika verification tool can automatically verify that Slang-based AADL component implementations conform to contracts, both at the code-level and model-level. Slang-based implementations of AADL systems can be executed directly or compiled to C for deployments on Linux or the seL4 verified microkernel.

ACM Reference Format:
John Hatcliff, Danielle Stewart, Jason Belt, Robby, and August Schwerdfeger. 2022. An AADL Contract Language Supporting Integrated Model- and Code-Level Verification. 1, 1, Article 1 (November 2022), 9 pages.
https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Over the last few decades, significant advancements in Model-Based Systems Engineering (MBSE) approaches, including modeling languages, model-level analyses, simulation, and code generation have improved the ability to design and deploy complex critical systems. Within the broader modeling space, the Architecture Analysis and Design Language (AADL) is distinguished by its relatively strong semantic emphasis (compared to other modeling languages like UML and SysML) and by its ecosystem of analysis tools that leverage that semantics. Tools that generate code from AADL models such as Ocarina [22], RAMSES [6], and HAMR [12] are helping fulfill the AADL community's MBSE visions by supporting the deployment of critical systems derived directly from models.

Due to in part to the AADL's strong semantics, researchers have developed formal model-based behavior specification and verification techniques. In particular, AADL component contract languages such as AGREE [8] and BLESS [21] have illustrated how system requirements can be refined into system and component-level formal specifications based on propositional and first-order logic.

Despite this progress, there are still gaps in the AADL MBSE vision related to deeply integrating behavioral specifications [15] at the model and code level. In recent years there has been significant progress on developing highly automated *code-level* verification tools including those for high-integrity languages such Spark Ada [16] and Frama-C [17], as well as program checking tools for general purpose languages including Java, C, C#, Rust, etc. However, both AGREE and BLESS analyses apply to the *model-level* and only address thread behavior specifications based on rather abstract notations (Lustre-based notations for AGREE, and state transition notations for BLESS). Given that the AADL has a significant vision related to code generation in the standard, including a code generation annex, descriptions of thread code organization (thread entry points), and descriptions of run-time libraries that implement foundational thread dispatching and communication steps, a stronger connection between model-level contract languages and code-level contract languages would be a significant step forward in further developing the AADL MBSE vision.

In this paper, we address these gaps by presenting an AADL contract language that can be utilized by AADL code generation to produce application source code with code-level contracts derived from model-level contracts. This required us to reorient concepts from both AGREE and BLESS to better align with notions of AADL thread entry points and AADL Application Programming Interfaces (APIs) associated with AADL's run-time services.

The contributions of this paper are as follows.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2022 Association for Computing Machinery.
XXXX-XXXX/2022/11-ART1 $15.00
https://doi.org/XXXXXXX.XXXXXXX

- We defined and implemented the AADL GUMBO[1] contract language (GCL) as an AADL annex, and we implemented full editing support for the GUMBO Contract Language (GCL) in the AADL OSATE IDE.
- We extended HAMR's multi-platform AADL code generator to automatically extract contracts from AADL models and weave them into the HAMR-generated code skeletons in the Slang high-integrity subset of Scala [25].
- We illustrated how the Logika-powered Sireum Integrated Verification Environment (IVE) for Slang can support contract reasoning at the code level, and that Logika can be used by engineers to verify that their thread implementations conform to contracts both at the code and model-level [20].

The framework described in this paper is included in the publicly available, open source High Assurance Modeling and Rapid engineering framework (HAMR) distribution [19]. A GitHub repository [18] provides the examples discussed in this paper, along with additional examples that illustrate the features of the contract language at both model- and source-code levels.

2 BACKGROUND CONCEPTS

AADL: SAE International standard AS5506C [5] defines the AADL core language for expressing the structure of embedded real-time systems via definitions of software and hardware components, their interfaces, and their communication. The AADL provides a precise, tool-independent, and standardized modeling vocabulary of common embedded software and hardware elements using a component-based approach [11]. The AADL standard also describes Run-Time Services (RTS) – a collection of run-time libraries that provide key aspects of threading and communication behavior. A major subset of the Run-Time Services has been formalized and a reference implementation has been developed [13], and we have designed our contract language and associated translation to code-level contracts with these definitions in mind.

HAMR: The HAMR framework generates code from AADL models for multiple execution platforms [12]. This includes generating threading, port communication, and scheduling infrastructure code that conforms to AADL run-time semantics as well as application code skeletons that engineers fill in to complete the behavior of the system. For the JVM platform, HAMR generates code in Slang [25], a high-integrity subset of Scala, which can be integrated with support code written in Scala and Java. Mixed Slang/Scala-based HAMR systems can also be translated to JavaScript (e.g., for simulation and prototyping) and run in a web browser or on the NodeJS platform. HAMR generates C infrastructure and application skeletons when targeting Linux and the seL4 micro-kernel [1]. Slang can be transpiled to C, and HAMR factors its C code through a Slang-based "reference implementation" of the AADL run-time and application code skeletons. Using the Logika verification framework for Slang (described below), Slang code can be verified with a high-degree of automation. This provides a basis for developing high-assurance AADL-based systems using Slang directly or via translation of Slang to C. C code transpiled from Slang can be compiled using standard C compilers, as well as the CompCert Verified C compiler [23].

Logika: Logika is a highly automated program verifier for Slang [20]. Slang's integrated contract language enables developers to formally specify method pre/post-conditions, data type invariants, and global invariants for global states. Verification of code conformance to contracts is performed compositionally and employs multiple back-end solvers in parallel, including Alt-Ergo [9], CVC4 [3], CVC5 [2], and Z3 [24]. The scalability of Logika is complemented by using incremental and parallel (distributable) verification algorithms. For situations where automated solvers cannot provide full verification, Slang includes an extensible proof language directly integrated with the programming language that Logika checks. Verification results, developer feedback on verification status, and contract/proof editing are supported in the Sireum Integrated Verification Environment (IVE) – a customization of the popular IntelliJ Integrated Development Environment (IDE).

The code-level contracts for the method include a *Requires* clause (pre-conditions), an *Ensures* clause (post-conditions), and a *Modifies* clause (frame conditions). Within the IVE, the developer can access program state/verification conditions and solver interactions.

The Logika verification engine uses asynchronous communications between the plugin client and tool server. This enables a seamless, on-the-fly integration similar to static type checking analysis usually offered by IDEs. Logika's main usability features include an as-you-type well-formedness analysis and verification of Slang programs by sending the checking tasks to a background server process, and visualizations of various helpful feedback propagated from the server as responses of the verification requests.

Example: This section presents a small example that we use for illustration.

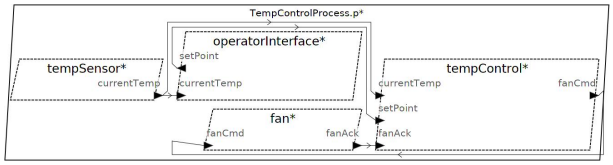

Fig. 1. Temperature Control Example – AADL Graphical View

Figure 1 presents the AADL graphical view for a simple temperature control system that maintains a temperature according to a set point containing high and low bounds for the target temperature. The periodic `tempSensor` thread measures the temperature and transmits the reading on its `currentTemp` data port. It sends a notification on its `tempChanged` event port if it detects that the temperature has changed since the last reading. When the sporadic (event-driven) `tempControl` thread receives a `tempChanged` event, it reads the value on its `currentTemp` data port and compares it with the most recent set point. If the current temperature exceeds the high set point or drops below the low set point, the fan is turned on or off respectively. In turn, the fan acknowledges these commands.

Typical Workflow: Given a collection of system requirements, AADL is used to develop a system architecture. As system requirements are refined to component-level requirements, the GCL is used to formally specify behavioral properties on component interfaces as well as system-level properties. Construction of the GCL specifications is interleaved with other AADL analyses for error modeling

[1]This material is based upon work supported by the U.S. Army Combat Capabilities Development Command, Aviation and Missile Center, under Contract No. W911W6-20-C-2020: Grand Unified Modeling of Behavioral Operators.

(hazard analysis), timing and schedulability analysis, and information flow analysis. Logika is used to verify compatibility of contracts at composition points as well as system model properties. When the architecture stabilizes, HAMR is used to generate the initial build infrastructure, AADL run-time code, and application code skeletons with model-level contracts translated down to code contracts. The IVE supports development of the AADL thread components using Slang, which also provides conventional unit and system testing. Logika is applied to verify that thread implementation conform to interface contracts. Verified Slang-based thread implementations can be executed with the HAMR AADL run-time on the JVM or translated to Javascript for simulation/visualization. As requirements and model-level GCL specifications change, HAMR can be re-run to update application code skeletons and contracts while preserving existing code. Slang implementations can be transpiled to C and deployed on Linux or the seL4 microkernel (to support strong spatial and temporal partitioning). The example used in this paper was completed using this workflow and deployed on the JVM, Javascript, Linux, and seL4. Similar HAMR workflows were used, e.g., on the DARPA CASE program for mission control software for military helicopters.

3 MODELING LANGUAGE ELEMENTS

There are four categories of contracts in the GCL: *integration constraints*, *data invariants*, *entry point contracts*, and *state variables*. Space constraints do not permit a detailed explanation of each, but important concepts of these categories are presented in Figure 2, which we refer to in the subsequent sections.

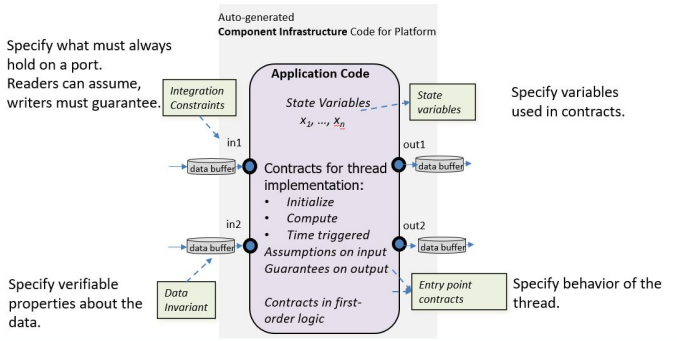

Fig. 2. Categories of contracts and how they fit into infrastructure code.

Integration Constraints: Integration constraints are the most simple conceptually and are likely the first to be used by engineers in typical workflows. The purple blocks in the concept graphic of Figure 2 indicate that each integration constraint pertains to a single port and specifies properties that must hold for any value passing through the port. They also enable engineers to specify constraints on port values that must be satisfied when components are integrated by connecting one port to another. The integration constraint shown below specifies a requirement on the usage of the `currentTemp` input data port of the temperature control thread: for any sender component S with output port p_o that is connected to (uses) the `currentTemp` port, the `degrees` field of all `Temperature.i` values flowing into the port must lie within the indicated range. `TC-Assume-01` is a unique label within the declaring component (`TempControl`) that is used for traceability. The optional string `"Current temperature range"` provides a longer human-readable description of the constraint that can be used in reporting. The `f32".."` is the current syntax within our prototype for specifying typed literals.[2]

```
integration
    assume TC-Assume-01 "Current temperature range":
        currentTemp.degrees >= f32"-70.0"
        & currentTemp.degrees <= f32"180.0";
```

The listing below shows an integration constraint on the `TempSensor` `currentTemp` output port. This specifies the valid operating temperature range for the sensor.

```
integration
    guarantee Sensor_Temperature_Range:
       currentTemp.degrees >= f32"-50.0"
       & currentTemp.degrees <= f32"150.0";
```

Intuitively, the integration of `TempSensor` to `TempControl` for the respective `currentTemp` ports is valid (i.e., the connected output and input ports are compatible) because any temperature value flowing out of the `TempSensor` that satisfies the `Sensor_Temperature_Range` constraint will satisfy the `TempControl` `TC-Assume-01` constraints.

In the underlying verification framework, the connection of any output port to input port gives rise to a verification obligation requiring evidence that any value flowing from a connected output port will satisfy any stated integration constraints on the input port. From a work flow perspective, this can be understood as part of the *component integration* activity. When the application logic of components are coded and verified as part of the *component development* activity, the verification framework requires evidence that all values placed in the output port by the code satisfy the output port integration constraints (i.e., `guarantee`). For input ports, the verification activity can `assume` any value flowing in from a component port will satisfy the integration constraints on the input port (since this must be guaranteed in the component integration activity as described above).

The concepts described above represent conventional "assume/guarantee" reasoning for component frameworks, and both AGREE and BLESS have analogous concepts. We have adopted the syntactic style of AGREE with the `assume` and `guarantee` keywords. For BLESS, integration constraints are a restricted form of BLESS port assertions. A key difference in the GCL is that integration constraints are distinguished from entry point contracts that specify input/output relationships – functional behavior of component implementations. These distinct concepts are handled using the same syntactic form in AGREE (assume/guarantee clauses) and BLESS (port assertions). Our rationale for having them in separate syntactic categories is explained in the discussion of entry point contracts.

Data Invariants: Requirements often specify invariants to ensure that data flowing through a system is well-formed — e.g., a temperature cannot be less than absolute zero, a timestamp that

[2]Ideally, the expression syntax used in contracts would be aligned with the envisioned AADL V3 expression language syntax. However, since the AADL V3 expression language and type system has not been developed yet, for simplicity of implementation, we have chosen to use Slang expressions and types in the contract grammar. Our implementation pipeline is designed to easily change the front-end concrete syntax for contracts as plans for the future of the AADL become more clear.

must be in 24-hour format. An invariant can also specify a well-formedness condition on a composite data structure. To support formal specification, what is needed in all these situations is the ability to associate a constraint representing an invariant with a datatype. The GCL supports the ability to add an invariant to any user-defined type specified using the Data Modeling Annex, a conventional notation in the AADL for defining data types.

The code snippet below shows a GCL invariant defined for the `SetPoint` data type. Bounds are defined for the `low` and `high` temperature values. Then a relational constraint specifies that the set point `degrees` of `low` is always less than or equal to that of `high`.

```
data SetPoint
  properties Data_Model::Data_Representation => Struct;
end SetPoint;
data implementation SetPoint.i
  subcomponents
    low: data TempSensor::Temperature.i;
    high: data TempSensor::Temperature.i;
  annex GUMBO {**
   invariants
     inv SetPoint_Data_Invariant:
      (low.degrees >= f32"50.0") &  (high.degrees <= f32"110.0")
      &  (low.degrees <= high.degrees);
  **};
end SetPoint.i;
```

In the underlying verification framework, everywhere a data value whose type has an associated invariant is passed or accessed, the fact that the value satisfies the invariant can be accepted as a premise. For this to be sound, at each point where a value of the type is created or updated, there is a verification obligation to show that the invariant holds. In relation to the other contract categories, whereas an integration constraint applies to a specific port, a data invariant on type T is relevant for any port whose type uses or includes type T. As indicated by Figure 2, the effective constraints on ports include both integration constraints and data invariants associated with types on the ports. For any such input port, the invariant can be assumed for values retrieved from the port; for any such output port, there is a verification obligation to show that the invariant holds before sending the data through the port.

AGREE and BLESS specification languages do not support the notion of a data invariant on Data Model Annex specifications. However, it seems straightforward to add this useful feature both to the specification languages and the underlying verification frameworks.

Entry Points Contracts: The AADL standard specifies that a thread's application code is organized into entry points as illustrated in Figure 2. The *Initialize* entry point is called once during the system's initialization phase and the *Compute* entry point is called repeatedly for the thread's normal dispatching.[3]

In addition to reading from and writing to ports, the thread application code may use local state variables whose values persist between entry point executions to perform computations and save previous port readings. Not every local state variable is relevant for a component's externally specified behavior. For those that are, the GCL provides a declaration clause to make the variables available for reference in entry point contracts (center of Figure 2). For example, a state variable `currentFanState` can be defined in the `tempControl` thread to store the most recent command sent to the fan:

```
state:
  currentFanState: CoolingFan::FanCmd;
  currentSetPoint: SetPoint.i;
  latestTemp: TempSensor::Temperature.i;
```

A thread's initialize entry point typically includes code to initialize thread local variables and to put initial values on output ports. The listing below shows excerpts of a GCL contract for the `TempControl` initialize entry point.

```
initialize
  modifies currentSetPoint, currentFanState, latestTemp;
  guarantee defaultSetPoint:
      (currentSetPoint.low.degrees == f32"70.0") &&  (
           currentSetPoint.high.degrees == f32"80.0");
  ...
```

A `modifies` clause is used to specify the variables that may be modified during entry point execution. A `guarantee` clause is used to specify properties that a variable value must satisfy at the completion of the entry point (similar clauses for `currentFanState` and `latestTemp` are omitted). In the contract for the `TempSensor` initialize entry point below, the value that the `currentTemp` output port must have at the completion of the entry point is specified.

```
initialize
   guarantee currentTempPortInitialVal:
       currentTemp.degrees == f32"72.0";
```

Initialize entry point contracts cannot have `assume` clauses since the purpose of the entry point is to initialize (no aspects of pre-state, including input ports or variable values, are allowed to be read).

A periodic thread component's compute entry point is invoked at intervals corresponding to the thread's declared period, whereas a sporadic thread's is invoked upon arrival of a message to event or event data input ports.[4] The GCL provides several contract variants for compute entry points. The most significant departure from AGREE or BLESS is that sporadic threads may have clauses that apply to all dispatches of the thread (i.e., they apply no matter what event triggers a dispatch), and then additional clauses can be added to specify behaviors that apply on the arrival of messages on specific ports. Excerpts from the `TempControl` compute entry point contract illustrate two clauses that constrain the post-state values of the thread's `latestTemp`, `currentSetPoint`, and `currentFanState` state variables regardless of whether the entry point is triggered by the arrival of, e.g., the `tempChanged` event or the `setPoint` message. (Note: in the listing below, the symbol `->:` is an implication.)

```
compute
    modifies currentSetPoint, currentFanState, latestTemp;

    guarantee  TC_Req_01:
          latestTemp.degrees < currentSetPoint.low.degrees
             ->: currentFanState == CoolingFan::FanCmd.Off;

    guarantee TC_Req_02:
          latestTemp.degrees > currentSetPoint.high.degrees
             ->: currentFanState == CoolingFan::FanCmd.On;
   ...
```

[3] AADL includes other entry points for finalization, performing mode changes, etc. These are not yet supported in HAMR code generation or our contract language.

[4] An AADL model can be configured to allow exceptions to this general rule, but we omit discussions of these special cases since they do not impact the design of our overall approach.

Using the GCL's handle clause, one can specify additional constraints that apply only when the thread is dispatched by the arrival of an event on a particular port. The contract excerpt below specifies that at the completion of the compute entry point when the thread is dispatched due a tempChanged event, the latestTemp local variable will be equal to the value of the currentTemp input data port at the time of dispatch.

```
handle tempChanged:
    modifies latestTemp;

    guarantee tempChanged:
       latestTemp == currentTemp;
```

The GCL also introduces a contract variant that supports case-based reasoning. Consider a variant of the temperature control system in which the TempControl thread is periodic with currentTemp and setPoint input data ports and a fanCmd output data port.

```
compute
  modifies latestFanCmd;
  cases
     case TC_Req_01:
        assume currentTemp.degrees < setPoint.low.degrees;
        guarantee (latestFanCmd == CoolingFan::FanCmd.Off)
                          & (fanCmd == CoolingFan::FanCmd.Off);
     case TC_Req_02:
        assume currentTemp.degrees > setPoint.high.degrees;
        guarantee (latestFanCmd == CoolingFan::FanCmd.On)
                          & (fanCmd == CoolingFan::FanCmd.On);
     ...
```

The contract excerpt above uses the cases clauses to specify, e.g., that when the currentTemp is less than the low set point at the time of dispatch, then at the completion of the entry point execution, the state variable latestFanCmd and the output port fanCmd are Off.

In the underlying semantics, due to the AADL dispatch semantics and notion of input port freezing, the compute entry point can be understood as a function from port input values and local state variables to output port values and possibly updated local state variables. Assume clauses place constraints on the input state, and guarantee clauses state constraints on the output state, sometimes referencing the input state. For example, guarantee clauses can reference the input values of state variables using an In(<varName>) construct as well as the values of input ports. The GCL entry point contracts can also include invariant clauses for state variables that hold for both initialize and compute entry points. Constructs are also available to reason about the absence or presence of events/messages on event and event-data ports. There are several minor well-formedness conditions that space constraints prevent us from enumerating completely. For example, assume clauses cannot appear in an Initialize entry point contract because there is no valid pre-state to refer to before the initialization occurs, and assume clauses cannot refer to output ports.

The GCL entry point contract notation was designed to be similar to AGREE, however, GCL includes a number of new concepts. AGREE was originally designed to support synchronous systems whose foundations were expressed in Lustre, and thus it does not have distinct initialize and compute contract forms aligned with AADL standard's entry point concepts. BLESS does not have distinct Initialize and Compute contracts due to its ties to behavioral specifications written in the AADL Behavioral Annex, which does not separate state machine behavior into separate entry points. The convenience notation for cases is not present in AGREE or BLESS.

4 MODEL CONTRACTS TO CODE CONTRACTS

4.1 Code Generation Overview

A detailed overview of HAMR code generation and how code for component application logic is integrated with HAMR's AADL runtime libraries is given in [12] (focusing on Slang code generation) and [4] (focusing on C code generation).

For each thread component, HAMR generates code that provides an execution context for a real-time task. This includes: (a) infrastructure code for linking application code to the platform's underlying scheduling framework, implementing storage associated with ports, and realizing the semantics associated with event and event-data ports, and (b) templates for developer-facing code including port APIs. Representations of the GCL contracts are woven into code (b).

To support semantic consistency for code generation across multiple languages and platforms, HAMR generates code in stages. A platform-independent implementation of the AADL Run-Time Services (RTS) is generated. HAMR specifies the API and aspects of RTS in Slang and then uses Slang extensions in Scala and C to implement platform-dependent aspects. As mentioned previously, there are multiple supported back-end targets including seL4 and Linux.

For the work in this paper, we focus on Slang. The GCL contracts are translated to Slang contracts, and Slang implementations of thread component application logic are verified to conform to the contracts. In addition to enabling verified Slang-based AADL system implementations, HAMR can already translate formally verified Slang-based component implementations to C, yielding high-assurance C-based deployments which can be compiled and executed on the seL4 formally verified microkernel. We believe that the same strategies that we use to inject representations of the GCL contracts into Slang could be used to inject contracts into the generated C code, e.g., for verification using a tool like Frama-C [17]. This could be used to provide further assurance of the Slang-to-C transpiler correctness. Alternatively, it can be used to support verification of HAMR systems in workflows that focused on writing C application code directly instead of coding at the Slang level.

At the front-end of the translation pipeline, we have extended the HAMR AADL Intermediate Representation (AIR) to include representations of the GCL contracts. This JSON-based representation enables us to support multiple modeling languages and environments. Currently, the GCL is implemented with full IDE support (type-checking, error reporting, code completion, etc.) as a plug-in for the AADL Eclipse-based Open Source AADL Tool Environment (OSATE) plug-in. Although the GCL is currently implemented as an AADL annex, it is external to the HAMR framework itself, and using the language-independent AIR gives us a mechanism to support multiple modeling languages, e.g., SysMLv2.

4.2 Translated Contracts

Integration Constraints: According to the principles introduced in Section 3, any value moving through a port must satisfy the integration constraints associated with the port. HAMR code generation generates dedicated APIs for putting and getting values to/from each

port. Thus, a natural strategy for realizing integration constraints at the code level is to add code contracts that: (a) require the value to be placed in an output port satisfies the associated constraints as pre-conditions to the put method, and (b) ensure the value to be retrieved from an input port satisfies the associated constraints as post-conditions.

For example, for the currentTemp input port in the TempControl thread, HAMR auto-generates: (a) a Slang representation of the predicate corresponding to the declared integration constraint, and (b) a contract on the get API that guarantees that the value retrieved satisfies the predicate.

```
// Auto-generated Slang predicate corresponding to
// currentTemp input port's Integration Constraint
@strictpure def currentTemp_ICPred(x:TempSensor.Temperature_i): B =
    -70.0f <= x.degrees & x.degrees <= 180.0f

// Auto-gen API for retrieving value from currentTemp input port
def get_currentTemp(): Option[TempSensor.Temperature_i] = {
  Contract(
    Ensures(currentTemp_ICPred(currentTemp),
            Res == Some(currentTemp))
  )
  ...(infrastructure code)...
  return value
}
```

For the corresponding sender side in the TempSensor, an analogous predicate is generated for the output port integration constraint, and the put method uses that in a pre-condition to ensure that all values placed on the port satisfy the constraint. Behind the scenes, the translation introduces Logika spec variables to provide a logical representation for the state of each port. Contract aspects that operate on ports (including the get and put methods above) use special spec clauses to read and update the port state abstractions during the flow of deductions in the verification.

```
// Auto-gen Slang predicate corresponding to
// currentTemp output port's Integration Constraint
@strictpure def currentTemp_ICPred(x:TempSensor.Temperature_i): B =
  x.degrees >= -50.0f & x.degrees <= 150.0f

// Auto-generated API for putting value on currentTemp output port
def put_currentTemp(value : TempSensor.Temperature_i) : Unit = {
  Contract(
    Requires(currentTemp_ICPred(value))
  )
  ...(infrastructure code)...
}
```

A key concept in this strategy is that the API methods serve as an abstraction for the underlying inter-component communication (specified via the AADL run-time services [13]). Verifying that the infrastructure code is correct *is not* the focus of the verification being presented here. Rather, when a HAMR back-end is developed for a new platform, there is an obligation within the overall assurance case to demonstrate that the platform implementation correctly implements the AADL run-time communication services. This assurance effort is carried out once, and then each application built using the platform relies on the previously developed assurance. This allows component developers and system integrators to focus on verifying that component implementations and their abstract connections conform to component and system level requirements (formalized in part, using the GCL contracts). We have other lines of work on formalizing the semantics of the AADL run-time services, establishing the conformance of platform implementations to those semantics, establishing the soundness of the contract framework verification conditions against the semantics, etc. Therefore, when the contract verification framework is applied to a HAMR code base, the implementations of the APIs presented above are not verified against their contracts. Instead, the contracts are used in the checking of client code (calls to the APIs) following the usual approach for dealing with library methods in contract verification frameworks.

The verification framework applied to the client code verifies that for calls to put methods, the argument (e.g., temp) satisfies the precondition of put which is derived from the port's integration constraint.

```
api.put_currentTemp(temp)
```

Correspondingly, in the receiver client code of TempControl thread,

```
latestTemp = api.get_currentTemp().get // .get always succeeds
// latestTemp inherits constraint assumptions
// from currentTemp port integration constraints
```

the return value from the get inherits constraints on the input currentTemp port based on the get post-condition (i.e., -70.0f <= tempControl.degrees and tempControl.degrees <= 180.0f).

To verify the compatibility of integration constraints of connected ports, a Slang script is generated that uses Slang proof constructs to establish the associated entailment. For example, for the connection above, the following script fragment states the desired integration property: for all values v of type Temperature_i, v satisfies integration constraint of the receiving port, under the assumption that it satisfies the integration constraint of the sending port. These proof scripts are verified by Logika "behind the scenes". More complicated forms of scripts are generated to reflect the composition of component entry points within a particular scheduling regime.

```
//  tempSensor.currentTemp --> tempControl.currentTemp
@pure def SensorCurrentTemp_TempControlCurrentTemp(
                                        v: Temperature_i): Unit = {
  Deduce(TempSensor_i_Api.currentTemp_ICPred(v) |-
         TempControl_i_Api.currentTemp_ICPred(v))
}
```

Data Invariants: HAMR translates each model-level datatype defined according to the AADL Data Model Annex into a Slang datatype. GCL datatype invariants are automatically translated to corresponding Slang datatypes using Slang's type invariant mechanism.

```
@datatype class SetPoint_i(val low: TempSensor.Temperature_i,
                           val high: TempSensor.Temperature_i) {
  // Slang datatype invariant
  @spec def SetPoint_Data_Invariant = Invariant(
    low.degrees >= 50.0f & high.degrees <= 110.0f &
      low.degrees <= high.degrees)                              }
```

Slang @datatype structures are immutable, and compiler optimizations allow them to be used efficiently for embedded code. In the Logika framework, each time the datatype constructor is used, there is a verification obligation to show that the supplied fields satisfy the invariant. Whenever the datatype is used, the verification framework can safely assume that the invariant holds. This achieves the GCL model-level semantics for datatype invariants.

Entry Point Contracts: Local state variables, state variable invariants, and all entry point contracts are automatically inserted into the thread component application code. Due to space constraints, we show only excerpts of the contract for the tempChanged handler of the TempControl thread.

```
def handle_tempChanged(api: TempControl_s_Operational_Api): Unit =
      {
  Contract(
    Modifies(
      // BEGIN COMPUTE MODIFIES tempChanged
      currentSetPoint, currentFanState, latestTemp,
      // END COMPUTE MODIFIES tempChanged
    ),
    Ensures(
      // BEGIN COMPUTE ENSURES tempChanged
      // guarantee TC_Req_01
      (latestTemp.degrees < currentSetPoint.low.degrees)
          ->: (currentFanState == CoolingFan.FanCmd.Off),
      // guarantee TC_Req_02
      (latestTemp.degrees > currentSetPoint.high.degrees)
          ->: (currentFanState == CoolingFan.FanCmd.On),
       ...
      latestTemp == api.currentTemp
      // END COMPUTE ENSURES tempChanged
    )
  )
  ...(user-supplied application logic)...
}
```

For sporadic threads, the HAMR structure for the Compute entry point is a collection of event handlers. The listing above shows that the GCL handler-independent `TempControl` compute `modifies` declarations and `guarantees` are injected directly into the Slang `handle_` method. Then the handler-specific constraint on `latestTemp` is added as appropriate for this handler. This last clause constrains the `latestTemp` state variable in the post-state to be equal to the Logika spec variable `api.currentTemp` representing the abstract state of the `currentTemp` input data port. There is an implicit conjunction for the `Ensures` clauses.

Contract Weaving: As the model goes through iterations of development, HAMR supports this by partial code-regeneration. One may regenerate into the same directory and HAMR will only make the changes in the code reflecting changes in the model and contracts. To avoid overwriting developer code upon contract changes at the model level, HAMR inserts comments to indicate where contracts begin and end (e.g., `// BEGIN INITIALIZES MODIFIES`). When code is regenerated, HAMR will parse the target code file, locate markers indicating contract blocks, and weave in updated contracts within the delimited regions. This supports iterative model and code development without the cost of overwriting developer-added application code in the thread implementations.

Code-level Verification: For our example system, Logika is able to verify that the Slang code for the thread entry points conforms to the contracts (including data invariants, etc.) in just a few seconds.

Figure 3 shows the code for the `tempControl` thread Initialize entry point and associated IVE verification annotations. The developer has filled in the implementation code, and the auto-generated contracts for the example are verified as evidenced by the *Logika Verified* banner at the bottom right of the figure. Logika incremental checking provides on-the-fly verification response as code is typed/edited. These capabilities were demonstrated in a video presentation (e.g., that shows checking of systems using a served-based parallelized checking using an 80-core server) at an industrial engagement event in January 2022 [5]. In addition to the example discussed in this paper, we have specified and verified contracts for a simple Isolette infant incubator medical device control system as well as high-integrity-oriented system components including voter-based sensor banks.

[5]Video available at https://bit.ly/tccoe22-logika

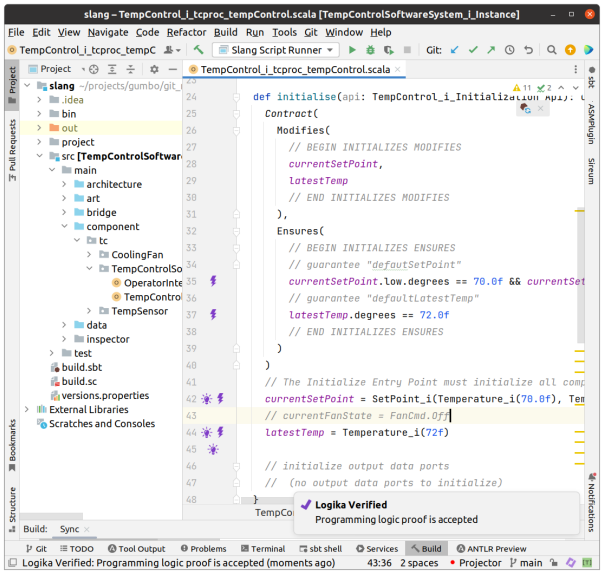

Fig. 3. Slang implementation of Initialize entry point code in Logika IVE

5 RELATED WORK

The most closely related works are the AGREE and BLESS AADL contract frameworks. AGREE originally targeted AADL models that emphasized dataflow with synchronous communication based on Lustre as an underlying computational model. Thus, AGREE most naturally handles thread components with data ports and with timing and state aspects aligned with Lustre. Component behaviors can be specified using equations relating inputs to outputs as well as more complex behaviors based on Lustre-inspired "node" blocks. Model-level verification is supported by the JKind model-checking framework, with the most common notions of verification being compatibility of guarantee/assume clauses on port connections, conformance of composed contracts on networks of sub-components to contracts on an enclosing component and verification of composed component behavior against equation-oriented property specifications. Recently AGREE has added support for events and static scheduling (similar to the strategy that we use, as needed to support seL4). In addition, "fold"-operations were added to support processing of inductively defined data types. AGREE specification and checking is supported by an OSATE plug-in that includes detailed reporting of verification status of claims and counter-examples. All fixed-width data types in AGREE are currently approximated using unbounded integers and reals. AGREE has had a strong influence within the AADL community, particularly in getting industrial users to understand the benefits of assume/guarantee specifications for component interfaces and the usefulness of automated formal methods. In our language design, we try to maintain these qualities while shifting the focus from the dataflow paradigm to the general tasking/communication primitives of AADL and to supporting integrated code level checking.

The BLESS contract language focuses on AADL thread components whose implementations are defined using an extension of AADL's Behavior Annex (BA) state transition notation. Typical verification activities include: (a) proving that a BLESS transition system for a thread (perhaps annotated with assertions) conforms

to its interface specification, and (b) proving that the composition of thread components satisfies end-to-end system properties. The BLESS OSATE plug-in supports editing of BLESS artifacts and coordination of verification activities. BLESS emphasizes a custom-built manual proof framework. Verification conditions are generated from interface specifications and transition systems, and then BLESS proof scripts apply proof rules to discharge the verification conditions. Building proof scripts requires significant additional effort on the part of developers compared with AGREE, but this approach supports more expressive specifications, verification of stronger properties, and detailed auditable evidence of property satisfaction. The more expressive specifications include rich notions of timing. Due to the ties to the AADL BA, compared to AGREE, BLESS more naturally supports AADL notions of event-based communication and complex event-oriented dispatch conditions. BLESS has been used to verify properties of complex embedded systems including pacemakers [21] and PCA infusion pumps [14]. GCL is less expressive than BLESS with respect to timing. In our language design, we have adopted many of BLESS's general verification condition principles and deductive structuring while emphasizing verification automation using automated solvers (using Slang manual proof steps only as necessary to help the automated solvers), and direct integration with source code verification and notions of AADL entry points.

6 CONCLUSION

We have presented a contract language for the AADL that supports integrated component contract specification and verification at both model and code levels. The model-to-code contract translation has been validated via an implementation within an industrial-relevant code generation framework. The feasibility of the verification strategy has been validated using Logika contract checking for the Slang high-integrity language.

Our contract language design builds on concepts from earlier AADL contract languages (AGREE and BLESS) and suggests directions for more closely aligning with code generation concepts called out in the AADL standard and AADL run-time service semantics [13]. While we have illustrated code-level concepts using Slang and Logika, we believe these same principles can be supported by other code-level verification frameworks that have been used in conjunction with AADL such as SPARK Ada [7] and Frama-C [17].

There are some limitations to our current implementation. First, the approach for entry point composition and end-to-end reasoning is specialized for the static scheduling approach [4] used in our most recent industrial project that emphasized system implementations for the seL4 verified microkernel. Second, the constructs for reasoning about AADL's event and event data ports apply to ports with buffer size one. While this aligns with buffer restrictions found from HAMR and other AADL code generation frameworks [22], moving beyond this restriction will broaden the applicability of the contracts to richer event-based systems that build on widely-used message-oriented middleware frameworks. We also hope to add greater support for timing-related specifications, aligned with AADL's standard timing properties and associated real-time scheduling frameworks [10].

Acknowledgments: The authors wish to thank Todd Carpenter for his valuable inputs on this research project and other GUMBO team members from Adventium Labs for the feedback they provided on the toolset.

REFERENCES

[1] 2015. seL4 Microkernel. sel4.systems/.
[2] Haniel Barbosa, Clark Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, et al. 2022. cvc5: a versatile and industrial-strength SMT solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 415–442.
[3] Clark Barrett, Christopher L Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. Cvc4. In *International Conference on Computer Aided Verification*. Springer, 171–177.
[4] Jason Belt, John Hatcliff, Robby, John Shackleton, Jim Carciofini, Todd Carpenter, Eric Mercer, Isaac Amundson, Junaid Babar, Darren Cofer, David Hardin, Karl Hoech, Konrad Slind, Ihor Kuz, and Kent Mcleod. 2022. Model-Driven Development for the seL4 Microkernel Using the HAMR Framework. *Journal of Systems Architecture* (2022), (to appear).
[5] SAE AS5506 Rev. C. 2017. Architecture Analysis and Design Language (AADL).
[6] Fabien Cadoret, Etienne Borde, Sébastien Gardoll, and Laurent Pautet. 2012. Design patterns for rule-based refinement of safety critical embedded systems models. In *2012 IEEE 17th International Conference on Engineering of Complex Computer Systems*. IEEE, 67–76.
[7] Bernard Carré and Jonathan Garnsworthy. 1990. SPARK—an annotated Ada subset for safety-critical programming. In *Proceedings of the conference on TRI-ADA'90*. 392–402.
[8] Darren D. Cofer, Andrew Gacek, Steven P. Miller, Michael W. Whalen, Brian LaValley, and Lui Sha. 2012. Compositional Verification of Architectural Models. In *Proceedings of the 4th NASA Formal Methods Symposium (NFM 2012)* (Norfolk, VA, USA), Alwyn E. Goodloe and Suzette Person (Eds.), Vol. 7226. Springer-Verlag, Berlin, Heidelberg, 126–140.
[9] Sylvain Conchon, Albin Coquereau, Mohamed Iguernlala, and Alain Mebsout. 2018. Alt-Ergo 2.2. In *SMT Workshop: International Workshop on Satisfiability Modulo Theories*.
[10] Rob Edman, Hazel Shackleton, John Shackleton, Tyler Smith, and Steve Vestal. 2015. A Framework for Compositional Timing Analysis of Embedded Computer Systems. In *IEEE International Conference on Embedded Software and Systems* (Newark, NJ).
[11] Peter H Feiler and David P Gluch. 2013. *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language*. Addison-Wesley. xx + 468 pages.
[12] John Hatcliff, Jason Belt, Robby, and Todd Carpenter. 2021. HAMR: An AADL Multi-platform Code Generation Toolset. In *Leveraging Applications of Formal Methods, Verification and Validation - 10th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2021, Rhodes, Greece, October 17-29, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 13036)*, Tiziana Margaria and Bernhard Steffen (Eds.). Springer, 274–295. https://doi.org/10.1007/978-3-030-89159-6_18
[13] John Hatcliff, Jerome Hugues, Danielle Stewart, and Lutz Wrage. 2022. Formalization of the AADL Run-Time Services. In *Leveraging Applications of Formal Methods, Verification and Validation - 11th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2022, Rhodes, Greece (To Appear)*.
[14] John Hatcliff, Brian R. Larson, Todd Carpenter, Paul L. Jones, Yi Zhang, and Joseph Jorgens. 2019. The open PCA pump project: an exemplar open source medical device as a community resource. *SIGBED Rev.* 16, 2 (2019), 8–13.
[15] John Hatcliff, Gary T. Leavens, K. Rustan M. Leino, Peter Müller, and Matthew J. Parkinson. 2012. Behavioral interface specification languages. *ACM Comput. Surv.* 44, 3 (2012), 16:1–16:58.
[16] Duc Hoang, Yannick Moy, Angela Wallenburg, and Roderick Chapman. 2015. SPARK 2014 and GNATprove. *International Journal on Software Tools for Technology Transfer* 17, 6 (2015).
[17] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. 2015. Frama-C, A Software Analysis Perspective. *Formal Aspects of Computing* 27, 3 (2015).
[18] SAnToS Laboratory. 2022. GCL case studies. https://github.com/santoslab/hilt22-case-studies/.
[19] SAnToS Laboratory. 2022. HAMR Project Website. https://hamr.sireum.org.
[20] SAnToS Laboratory. 2022. Sireum Logika. https://logika.v3.sireum.org/index.html.
[21] Brian Larson, Patrice Chalin, and John Hatcliff. 2013. BLESS: Formal Specification and Verification of Behaviors for Embedded Systems with Software. In *Proceedings of the 2013 NASA Formal Methods Conference (Lecture Notes in Computer Science,*

Vol. 7871). Springer-Verlag, Berlin Heidelberg, 276–290.

[22] Gilles Lasnier, Bechir Zalila, Laurent Pautet, and Jérome Hugues. 2009. Ocarina: An environment for AADL models analysis and automatic code generation for high integrity applications. In *International Conference on Reliable Software Technologies*. Springer, 237–250.

[23] Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. 2016. CompCert-a formally verified optimizing compiler. In *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*.

[24] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

[25] Robby and John Hatcliff. 2021. Slang: The Sireum Programming Language. In *International Symposium on Leveraging Applications of Formal Methods*. Springer, 253–273.